Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Navegación de Robots Móviles en entorno Matlab-ROS

Autor: SANDRA CARRASCO LIMEROS

Tutor: ELENA LÓPEZ GUILLÉN

2019

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Navegación de Robots Móviles en entorno Matlab-ROS

Autor: SANDRA CARRASCO LIMEROS

Director: ELENA LÓPEZ GUILLÉN

Tribunal:

Presidente: IGNACIO BRAVO MUÑOZ

Vocal 1º: CÉSAR MATAIX GÓMEZ

Vocal 2º: ELENA LÓPEZ GUILLÉN

Calificación: .....

Fecha: .....

"Cuanto más grande es la prueba, más glorioso es el triunfo."
William Shakespeare

Agradecimientos

Tanto si crees que puedes, como si no, en ambos casos estás en lo cierto.

Henry Ford

La realización de este TFG supone el fin de una etapa muy importante en mi vida, la cual me ha hecho crecer como persona, tanto en lo profesional como en lo personal.

En primer lugar, me gustaría agradecer a mi tutora, Elena López, por su acompañamiento, su energía y su apoyo durante toda la realización de mi TFG.

A mis padres y a toda mi familia, gracias a quienes soy quien soy y hacia quienes sólo puedo expresar mi agradecimiento por apoyarme durante la etapa académica que hoy culmina. Gracias por apoyarme en todas mis decisiones y ayudarme a lograr mis metas.

Y, por último, pero no menos importante, a mis amigos, por acompañarme todos estos años y ayudarme a atravesar los momentos más difíciles.

Resumen

En el presente documento se describe el proceso llevado a cabo para desarrollar una serie de aplicaciones relacionadas con la navegación autónoma de robots móviles destinadas al ámbito docente y de investigación, utilizando como entorno la Robotics System Toolbox de Matlab y ROS. Estas incluyen algoritmos de: mapeado y SLAM, localización (AMCL), evitación de obstáculos (VFH), seguimiento de pasillos usando lógica difusa y la transformada de Hough, y planificación global (PRM).

Además, se diseña un aplicación para el guiado de vehículos autónomos en el simulador CARLA, con la cual se participa en el primer CARLA Challenge, celebrado entre abril y junio del 2019.

Palabras clave: navegación autónoma, ROS, Robotic System Toolbox, SLAM, mapeado, gmapping, AMCL, PRM, VFH, localización, planificación, spline, CARLA Challenge.

Abstract

The main aim of this project is to develope a set of algorithms related to autonomous navigation in the academic and research fields. They were implemented in a ROS-Matlab environment, using the Robotics System Toolbox framework. The algorithms cover a wide range of applications: mapping and SLAM, localization (AMCL), obstacle avoidance (VFH), hallway tracking using fuzzy logic and Hough Transform, and global planning (PRM).

Furthermore, an aditional goal was set. The design and implementation of an application for an automated guided vehicle in CARLA simulator, with the participation in the first CARLA Challenge.

Keywords: autonomous navigation, ROS, Robotic System Toolbox, SLAM, mapping, gmapping, AMCL, PRM, VFH, localization, planning, spline, CARLA Challenge.

Resumen extendido

Entre los movimientos y las tendencias en el ámbito educativo que pretenden conferir al alumnado capacidades para afrontar los retos sociales y profesionales del siglo XXI, la robótica se encuentra entre los más destacados. La transversalidad inherente a la práctica robótica, sustanciada a través de la combinación de diversas técnicas y tecnologías, hacen de ella una disciplina que otorga las habilidades necesarias para la adquisición de una de las competencias clave más importantes: aprender a aprender.

Dada la importancia que está adquiriendo esta disciplina, surge la idea del presente trabajo, cuya finalidad es el desarrollo de aplicaciones de navegación para robots móviles utilizando un framework concreto basado en la nueva *Robotics System Toolbox* de MATLAB que año tras año incorpora nuevas funcionalidades y algoritmos. Este desarrollo se realizará así mismo junto con un meta-sistema operativo para aplicaciones de robótica, ROS (*Robot Operating System*), que permite su implementación sobre diferentes simuladores y plataformas reales. El objetivo es llegar a tener un conjunto de aplicaciones de navegación para el ámbito docente y de investigación dentro del Departamento de Electrónica. Se investiga así la fiabilidad y eficiencia de estos algoritmos en simulación y en dos plataformas robóticas reales: el Amigobot y el Seekur.

Estas aplicaciones incluyen: mapeado y SLAM, donde se hace una comparativa entre tres algoritmos distintos, localización con el Método de Monte Carlo Adaptativo (AMCL), evitación de obstáculos con el algoritmo Vector Field Histogram (VFH), seguimiento de pasillos utilizando lógica borrosa y la transormada de Hough, y planificación global con el algoritmo Probabilistic Roadmap (PRM) acompañado de un controlador de tipo Pure Pursuit.

De esta manera se pueden comprobar las diferencias en funcionamiento y las dificultades a afrontar cuando se pasa de un entorno simulado a una plataforma robótica real, así como las grandes diferencias que existen entre distintas plataformas.

Para llevar a cabo este cometido, surge la necesidad de cumplir una serie de objetivos específicos. Primero se realiza un análisis del entorno de desarrollo, tanto de ROS como de la Robotic System Toolbox de Matlab. Tras esto, se estudia y configura la arquitectura del sistema tanto en simulación como con las plataformas robóticas reales, estableciendo la comunicación entre las distintas máquinas. Conociendo la arquitectura del sistema y tras comprobar el correcto funcionamiento se procede al desarrollo e implementación individual de los distintos algoritmos citados anteriormente.

Una vez implementados todos los algoritmos individualmente, se crea una aplicación completa donde el robot recorre el mapa evitando obstáculos (VFH) hasta que se localiza en este (AMCL),

y crea una ruta hasta el destino (PRM) al que llega mediante un controlador *Pure Pursuit*.

Con este detenido estudio de la Robotic System Toolbox de Matlab y su implementación en un entorno Matlab-ROS, se podrán utilizar estos algoritmos para la creación de prácticas de laboratorio en asignaturas como *Robótica Móvil*, dentro del Máster de Ingeniería Industrial.

Como segunda parte de este trabajo, se explora un campo muy novedoso relacionado con la navegación autónoma: el guiado de vehículos autónomos.

Para ello, se hace uso del novedoso simulador CARLA. Con este simulador se llevará a cabo la creación y depuración de una aplicación de guiado de vehículos utilizando el entorno de desarrollo de aplicaciones robóticas ROS. La motivación principal es la participación en el primer CARLA Challenge, organizado por un equipo compuesto por varios investigadores del *Computer Vision Center* de la Universidad de Barcelona (UAB), laboratorios Intel y el *Toyota Research Institute*, y patrocinado por grandes empresas como *AWS*, WAYMO, AUDI, Volkswagen, Uber, EvalAI y Alpha Drive entre otras. Este concurso presenta una serie de escenarios urbanos con distintos retos a superar, como evitar ciclistas, peatones y otros obstáculos, cumplimiento de semáforos y señales, etc.

El desarrollo de esta aplicación se puede dividir en cuatro fases. Primero se codifica la generación de trayectorias a partir de waypoints mediante una interpolación de tipo spline. Tras esto se programa un controlador óptimo LQR con predicción para el seguimiento de la trayectoria. Dada la necesidad de detectar semáforos y señales, se hace uso de una pequeña aplicación basada en la red neuronal pre-entrenada YOLO y, tras esto, se configura un comportamiento reactivo en función de los estímulos percibidos. Por último, adecuamos la aplicación a la estructura requerida en el Challenge, basada en un agente de Python, el cual será enviado al concurso como una imagen docker.

Todo lo descrito anteriormente se ha estructurado en una serie de capítulos y secciones en los que se detallan y explican cada uno de los pasos llevados a cabo para la realización del presente proyecto.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Desocupado lector, sin juramento me podrás creer
que quisiera que este libro [...] fuera el más
hermoso, el más gallardo y más discreto que
pudiera imaginarse.

Miguel de Cervantes, Don Quijote de la Mancha

La finalidad principal de este trabajo es el desarrollo de aplicaciones de navegación para robots móviles utilizando un framework concreto basado en la nueva *Robotics System Toolbox* de MATLAB que año tras año incorpora nuevas funcionalidades y algoritmos. Este desarrollo se realizará así mismo junto con un meta-sistema operativo para aplicaciones de robótica, ROS (*Robot Operating System*), que permite su implementación sobre diferentes simuladores y plataformas reales. El objetivo es llegar a tener un conjunto de aplicaciones de navegación para el ámbito docente y de investigación dentro del Departamento de Electrónica.

Con el fin de comprender la temática del presente trabajo, se describe a continuación una visión general de los fundamentos asociados a la teoría básica de navegación y posicionamiento en robots móviles, así como la situación del estado del arte en este área y la estructura y objetivos seguidos en la realización del trabajo.

1.1 Navegación autónoma de robots móviles

Este trabajo se centra en la navegación autónoma de robots móviles. Dado un conocimiento parcial sobre su entorno y una posición de destino, la navegación abarca la capacidad del robot para actuar basada en su conocimiento y en las medidas de los sensores para alcanzar su objetivo de manera tan eficiente y fiable como sea posible.

La navegación es uno de los mayores retos requeridos en robots móviles. Alcanzar el éxito en la navegación requiere el éxito en los cuatro bloques que comprende: percepción, el robot debe interpretar los sensores para extraer datos útiles; localización, el robot debe determinar su posición en el entorno; cognición, el robot debe decidir cómo actuar para alcanzar su meta; y control, el robot debe modular sus motores para alcanzar la trayectoria deseada.

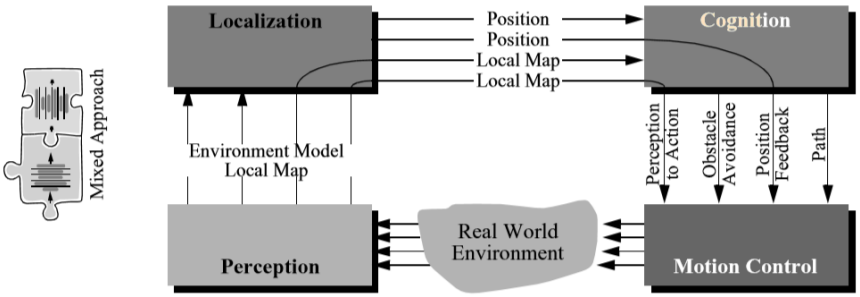

Figura 1.1: Arquitectura del problema de la navegación autónoma.

De estos cuatro componentes, la localización ha recibido la mayor atención dentro de la comunidad investigadora en la última década, dado que, para prácticamente cualquier tarea que el robot deba realizar, se ha de determinar su posición y orientación en el espacio con un cierto nivel de precisión. Además es uno de los problemas más desafiantes, debido a que la información de la velocidad y posición del robot se obtiene de sensores con ruido y no-linealidades, aspectos que, si no son tomados en cuenta, pueden llevar a un crecimiento ilimitado (no acotado) de la incertidumbre de estimación, con lo cual el robot no sabría con certeza su posición y no podría realizar adecuadamente sus tareas. Como resultado, se han logrado avances significativos en este área.

En la figura 1.2 se muestra un esquema general del proceso de localización, en el que se observa cómo este es iterativo.

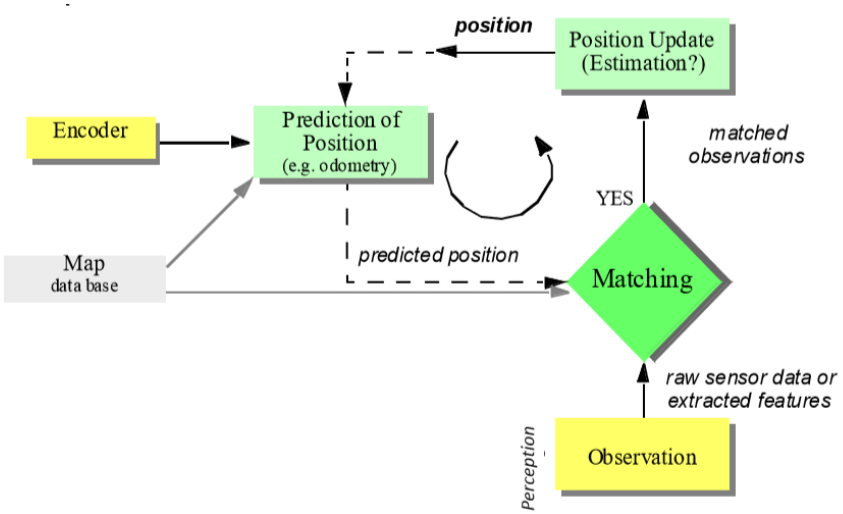

Figura 1.2: Esquema general del proceso de localización en robots móviles. [1]

La localización del robot en el instante k, (x_k, y_k, θ_k) se puede estudiar de varias formas. Suponiendo que la posición inicial antes de iniciar el movimiento es conocida, la posición actual del robot se puede estimar usando información local del movimiento obtenido a través de distintos sensores, de forma que se calcule la distancia recorrida desde el punto inicial; a esto se le conoce como *odometría*. Esta estima la posición del robot en un plano a partir de la posición en el instante anterior *k-1* y la velocidad del robot v en los ejes (x, y) junto con la velocidad angular, w. Para un periodo de muestreo Ts pequeño se tiene que la posición en el instante

actual, k, está dada por la ecuación:

$$L = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + T_s \begin{bmatrix} v_{k-1} cos\theta_{k-1} \\ v_{k-1} sin\theta_{k-1} \\ w_{k-1} \end{bmatrix}$$

En caso de que la posición inicial del robot sea desconocida, pero este disponga de un sensor que pueda determinar la posición relativa del robot a puntos de referencia (Landmarks) no preestablecidos que observa y extrae del entorno, entonces el robot podrá construir un mapa de este y obtener su posición absoluta en dicho mapa de forma simultánea utilizando la fusión de la información proveniente de los sensores de movimiento y los sensores del entorno. Este procedimiento se conoce como localización y mapeado simultáneo (SLAM [2] [3] [4] [5]).

Si el robot dispone previamente de un mapa del entorno, el mismo tipo de sensor puede utilizarse para determinar la posición del robot en el mapa, al comparar las características observadas en el entorno con la información almacenada en memoria.

Dado que la información de la posición del robot se necesita en el algoritmo de navegación, el tiempo de respuesta del método de localización resulta muy relevante. Si se pierde la información de la localización del robot o se recibe de forma lenta, el algoritmo de control no tendrá actualizados los datos y la acción de control producida será incorrecta, lo cual podría provocar la desestabilización del sistema.

Dentro de los bloques de cognición y control encontramos dos competencias claramente diferenciadas. Dado un mapa y una ubicación destino, la planificación de la ruta implica identificar una trayectoria que hará que el robot alcance dicha ubicación. La planificación de trayectoria es una competencia estratégica para resolver problemas, ya que el robot debe decidir qué hacer a largo plazo para logar sus objetivos. La segunda competencia es igualmente importante, pero ocupa el extremo táctico opuesto. Dadas las lecturas del sensor en tiempo real, la evitación de obstáculos significa modular la trayectoria del robot para evitar colisiones. Una gran variedad de enfoques que se verán más adelante han demostrado ser competentes para evitar obstáculos.

En inteligencia artificial, la planificación y reacción se consideran a menudo como enfoques contrarios o incluso opuestos. De hecho, cuando se aplican a sistemas físicos como robots móviles, la planificación y la reacción tienen una fuerte complementariedad, siendo cada uno crítico para el éxito del otro. El desafío de la navegación para un robot consiste en ejecutar un plan de acción para alcanzar su meta. Durante la ejecución, el robot debe reaccionar ante eventos imprevistos sin que esto le impida alcanzar su meta. Sin reaccionar ante imprevistos, el esfuerzo de planificación no sería rentable porque el robot nunca alcanzaría físicamente su objetivo. Así mismo, sin planificación, la reacción por sí sola no puede guiar el comportamiento general del robot para alcanzar un objetivo.

1.2 Estado del Arte

En este apartado se realiza una revisión del estado del arte dentro del contexto del presente trabajo. Se han revisado numerosas investigaciones relacionadas con él y se han considerado los diversos enfoques dentro de cada disciplina.

1.2.1 Métodos de Mapeado

En mapeado, la elección del método de representación dependerá de la aplicación, las características observables y la carga computacional que requiera dicho método. Podemos diferenciar entre dos tipos de representaciones: continuas (muy precisas pero con alta carga computacional) y discretas (mucho más simplificadas) [26].

Como representación continua tenemos los mapas geométricos basados en la arquitectura del entorno. Se trata de una representación con un conjunto de líneas infinitas, por lo que requieren una elevada carga computacional.

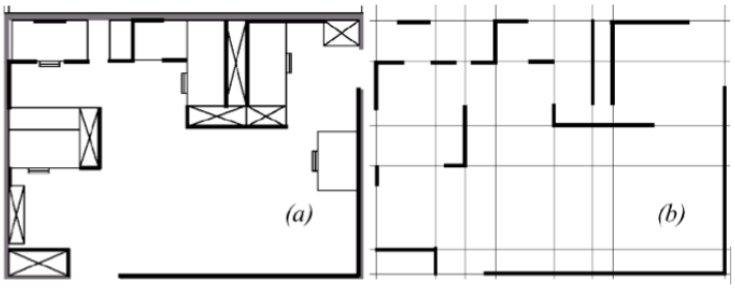

Figura 1.3: Mapa geométrico.

Dentro de las representaciones discretas, mucho más extendidas, encontramos distintos métodos:

- `Descomposición en celdas exactas`. Se busca cubrir el espacio no ocupado mediante teselas o polígonos. Lo importante es la capacidad que tiene el robot de pasar de un área libre a otra almacenando las relaciones entre las diferentes áreas en los llamados diagramas de conectividad.

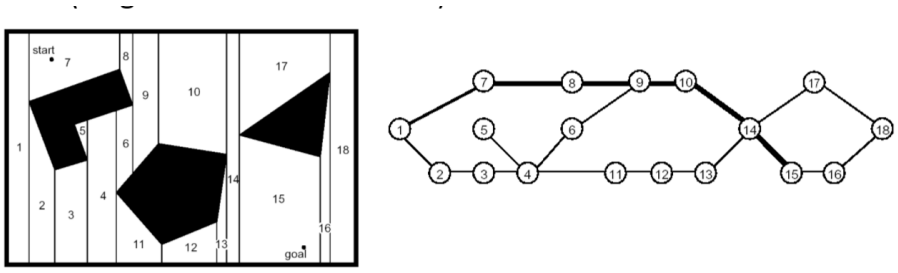

Figura 1.4: Descomposición en celdas exactas.

- `Descomposición en celdas fijas`. Se descompone el espacio en celdas fijas con un valor binario: ocupada (1) o libre (0), siendo posible presentar un estado desconocido. El problema de esta representación es que desaparecen los pasos estrechos.

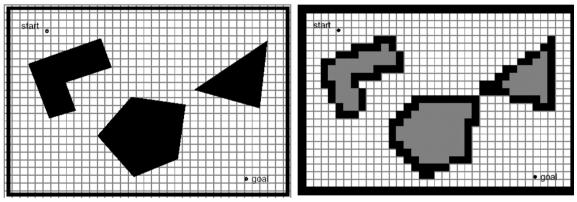

Figura 1.5: Descomposición en celdas fijas.

- `Descomposición en celdas de tamaño variable`. Con este tipo de representación se resuelve el problema de los pasos estrechos. Se parte de un tamaño fijo que es el que se mantiene en las zonas libres, de manera que si queda una zona ocupada en la celda, se divide en cuatro y así sucesivamente hasta una resolución máxima especificada.

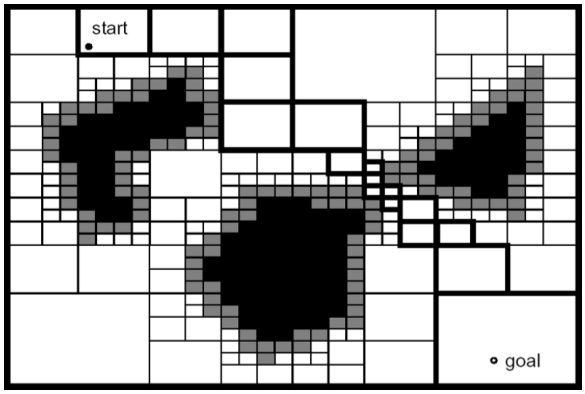

Figura 1.6: Descomposición en celdas de tamaño variable.

- `Rejillas de ocupación`. Se utilizan celdas de un tamaño mínimo y a cada una de ellas se le asigna un contador que se incrementa con cada impacto de los sensores de distancia (como el láser) y se decrementa con un impacto en una celda oculta tras ella. El problema de este método es que los mapas crecen a medida que lo hace el entorno.

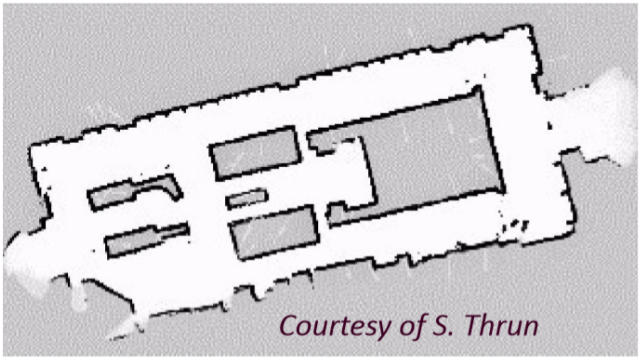

Figura 1.7: Rejillas de ocupación.

- `Representación topológica`. En este caso evitamos las medidas geométricas del entorno, concentrándonos en características relevantes que son útiles para los objetivos del robot. Tenemos así nodos, utilizados para denotar áreas de interés, y arcos, utilizados para indicar la adyacencia de dos nodos. Cuando un arco conecta dos nodos, indica que se puede pasar de un nodo a otro sin necesidad de atravesar otro nodo. A diferencia de la discretización basada en celdas, aquí los nodos no tienen por qué estar separados un tamaño fijo.

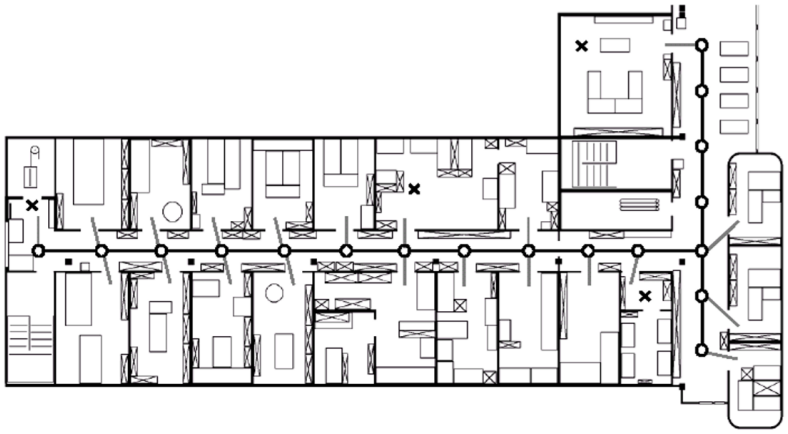

Figura 1.8: Representación topológica.

A la hora de construir un mapa, este tiene que tener la capacidad de añadir nuevos elementos detectados. Así mismo, debe presentar información para estimar la posición del robot y para planificar y realizar otras tareas de navegación, como la evitación de obstáculos. Para medir la calidad del mapa se puede realizar una corrección topológica o una corrección métrica. El principal problema del mapeado es que la mayoría de los entornos tienen una mezcla de características predecibles y no predecibles. Se debe mantener una consistencia del mapa ante cambios.

Actualmente, como se explicará en el apartado de localización 1.2.2, la técnica más empleada es SLAM, *Simultaneous Localization and Mapping* [2] [3] [4] [5] la cual consiste en que el robot, partiendo de un punto cualquiera, debe ser capaz de explorar autónomamente el entorno utilizando sus sensores y construir el mapa a la vez que se localiza sobre él. El problema de esta técnica es que el error cometido en la localización se traslada al mapa y viceversa. Los pequeños errores se van acumulando llegando a producir graves errores globales, sobre todo en los cierres de lazos.

Actualmente, tanto para mapeado como para localización los métodos más usados son los métodos probabilísticos [28][29]. Estos buscan lidiar con el problema de la incertidumbre en el entorno, el robot, los sensores y los modelos, consiguiendo sistemas robóticos robustos. Introducen información sobre las ambigüedades debidas a los modelos y a los sensores, trabajando con distribuciones de probabilidad en lugar de con datos exactos.

Estos métodos se basan en el teorema de Bayes, que permite calcular probabilidades que de otro modo son difíciles de obtener.

Bajo el supuesto de Markov, [22], la actualización recursiva de la fórmula de Bayes permite integrar eficientemente múltiples condiciones para la estimación del estado.

Se definen así los filtros Bayesianos, ampliamente usados en robótica, no solo en percepción, si no también en planificación local y global, como se explicará más adelante en los apartados 1.2.3 y 1.2.4. Estos definen para cada sistema un conjunto de estados S, de acciones A que hacen pasar de un estado a otro y observaciones O, que se pueden obtener en los diferentes estados. El sistema es dinámico, de manera que a lo largo del tiempo se van sucediendo las acciones y observaciones. Además se conocen dos modelos probabilísticos del sistema: el modelo de actuación, para caracterizas las incertidumbres de las acciones p(s'|s,a); y el modelo de percepción, para caracterizar las incertidumbres de las observaciones p(o|s).

El objetivo del filtro Bayesiano es estimar el estado del sistema en cada instante de tiempo, manteniendo para ello una distribución de probabilidad sobre la variable S. Esto es conocido como distribución de creencia o Bel(S) y caracteriza, para cada valor particular del estado, su probabilidad de ser el estado real del sistema. Esta distribución será uniforme inicialmente si el estado inicial es desconocido, o una delta si es conocido.

Se debe cumplir la propiedad de Markov:

$$\boxed{p(s_{t+1}|s_t, d_t) = p(s_{t+1}|s_t, a_t)} \tag{1.1}$$

Esta equivale a decir que conocer el estado presente hace que el futuro sea independiente del pasado, de manera que el estado actual, junto con la acción actual, son suficientes para estimar el siguiente estado. Un filtro de Bayes puede aplicarse en dos etapas:

- Una primera etapa de predicción, tras la ejecución de una acción a en el instante *t-1*, en el que se aplica el modelo de actuación:

$$\boxed{Bel_t(s') = \int_s p(s'|s, a) Bel_{t-1}(s-1)} \tag{1.2}$$

- Una segunda etapa de estimación, tras la obtención de una observación o en el nuevo estado, usando el modelo de percepción.

$$\boxed{Bel_{t+1}(s) = \eta p(o|s) Bel_t(s)} \tag{1.3}$$

Ahora se va a analizar la aplicación de estos filtros bayesianos a la obtención de mapas concretamente, suponiendo que la posición del robot es conocida sin errores. El mapeado no es una tarea sencilla, ya que hay distintos problemas que se derivan de él: el tamaño del mapa respecto al rango de percepción sensorial del robot, el ruido en la percepción y la actuación, el solapamiento perceptual (lugares distintos que se ven igual por los sensores) y la detección de lazos (regresar al mismo punto por un camino distinto).

Para ello se usan las rejillas de ocupación, que se basan en representar el entorno mediante una rejilla y estimar la probabilidad de que cada celda esté ocupada por un obstáculo. Se supone que la posición del robot es conocida y que la ocupación de cada celda es independiente de las demás. La probabilidad de cada celda se va actualizando utilizando un filtro de Bayes binario:

$$Bel(m_t) = \eta p(z_t|m_t) \int p(m_t|m_{t-1}, X_{t-1}) Bel(m_{t-1}) dm_{t-1} \tag{1.4}$$

El mapa se actualiza utilizando un modelo inverso del sensor:

$$Bel(m_t) = 1 - (1 + \frac{P(m_t|z_t,u_{t-1})}{1-P(m_t|z_t,u_{t-1})} \frac{1-P(m_t)}{P(m_t)} \frac{Bel(m_{t-1})}{1-Bel(m_{t-1})}) \tag{1.5}$$

Pero, en la practica, el robot no conoce su posición exacta mientras se obtiene el mapa. Se plantea el problema de que para obtener el mapa se requiere la posición, y para obtener la posición se requiere el mapa. De aquí es de donde surge la idea del SLAM previamente mencionada, en el que dadas las acciones y las observaciones se estima la posición de las marcas del mapa y el camino seguido por el robot. Diferenciamos entre Full Slam, en el que se estima el mapa y la trayectoria completa del robot y Online Slam, en el que se estima el mapa y la posición actual del robot.

En el siguiente apartado se definirá más en detalle el concepto de SLAM, así como la aplicación de métodos probabilísticos en localización.

1.2.2 Métodos de Localización

La localización de robots en interiores se considera esencial para que un robot móvil alcance la autonomía. Con esta información crítica, el robot móvil puede interactuar de manera coherente con objetos y personas del entorno y puede navegar a través de los objetos de alrededor con flexibilidad para enfrentarse a situaciones inesperadas. Algunas de estas aplicaciones son exploración, búsqueda y rescate, vigilancia, recuperación, transporte y otras misiones.

La localización se puede subdividir en dos problemas claramente diferenciados. El posicionamiento global es la habilidad de determinar la posición del robot en un mapa conocido a priori, sin conocer su posición inicial. Si no se conoce el mapa a priori, muchas aplicaciones permiten que este se construya a la vez que el robot explora el entorno. Una vez que ha sido localizado en el mapa, el seguimiento local define el problema de mantener un seguimiento de la posición del robot en el tiempo.

En esta sección se repasarán las técnicas utilizadas hasta el momento en localización de robots en interiores. Primero se hará un repaso de los sensores utilizados para dicho fin, y se proseguirá con un estudio de los algoritmos desarrollados en los últimos años.

1.2.2.1 Sensores utilizados en Localización.

Existen diversos métodos, en cuanto a hardware, para conseguir la localización en interiores, dependiendo del entorno, aplicación y disponibilidad.

- Las técnicas basadas en `radio frecuencia`, [10], han sido muy populares últimamente entre los investigadores. Este se trata de un sistema de almacenamiento y recuperación de datos remoto que usa dispositivos denominados etiquetas, tarjetas o transpondedores

RFID. El propósito fundamental de este tecnología es transmitir la identidad de un objeto (similar a un número de serie único) mediante ondas de radio.

- El `Sistema de Posicionamiento Global` o GPS, [11] [12], es uno de los métodos usados para la indicación de posición. El GPS depende de satélites que orbitan la Tierra y transmiten señales de radio de tiempo precisas. Proporciona una estimación de posición tridimensional en coordenadas absolutas comparando el retardo de tiempo de diferentes señales de satélite. Hoy en día, los GPS más precisos se ubican dentro de un promedio de 2.5m de error circular probable (CEP), lo que no es suficiente para una aplicación precisa de localización de robots. Además, la señal del GPS tiende a desvanecerse más rápido dentro de edificios, por lo que se usa más en aplicaciones exteriores.

- La localización en interiores basada en `GSP` utiliza los módulos GSP conectados al objetivo en movimiento. El módulo GSP se comunica con la estación base más cercana y calcula la posición probabilística del objeto. El uso de la tecnología GSM tiene ventajas frente a otras técnicas. Primero, un sistema basado en la señal móvil tiene la ventaja de utilizar el hardware existente del teléfono sin la necesidad de interfaces de radio adicionales. La cobertura y capacidad GSP superan la de otras tecnologías. Además, un sistema de localización basado en móvil es robusto frente a fallos de energía en interiores.

- El `infrarrojo`, [13], es una radiación de energía con una frecuencia por debajo de la sensibilidad del ojo humano. Basado en la señal reflejada, el sensor infrarrojo podría determinar la ubicación del objeto. Se utilizan múltiples transmisores y receptores de infrarrojos para la localización.

- El propósito de la técnica de `ultrasonido`, [14], es medir la distancia entre un punto fijo y un objeto móvil usando ondas ultrasónicas. El sistema debe consistir en un trasmisor y varios receptores. Para sincronizar los receptores, se usa una onda más rápida que la ultrasónica, como la radio o infrarrojo. Una vez que los receptores detectan las ondas ultrasónicas reflejadas, calculan el tiempo entre la señal de sincronización y la señal ultrasónica reflejada para estimar la distancia entre el transmisor y el objetivo. La utilidad de esta técnica es la simplicidad de implementación y la reducción de costos. La principal desventaja de los sensores de ultrasonido es la trayectoria múltiple generada en el receptor, que podría perturbar la distancia entre el emisor y el objetivo.

- La técnica de la red de área local inalámbrica o `WLAN`, [15], requiere información sobre el tiempo de viaje de la onda de radio o la intensidad de la señal recibida para estimar la ubicación. El uso de la WLAN para localización tiene unos beneficios fundamentales. Primero, la disponibilidad de múltiples puntos de acceso WLAN en áreas urbanas hace que sea fácil de explotar para propósitos de posicionamiento. En segundo lugar, el enfoque WLAN da lugar a errores de posición limitados, no acumulativos. En tercer lugar, alcanza la señal de GPS en áreas cerradas. Finalmente, se requiere flexibilidad de ubicación y puertos de datos. Con la ausencia de línea de visión y la presencia de pérdida de ruta, diversidad de rutas y desvanecimiento, la información de la señal se vuelve inexacta y errónea, lo que se considera un principal inconveniente de WLAN.

- Existen otras tecnologías físicas utilizadas para fines de localización en interiores. La utilización de un odómetro, [16], o encoder, [17] es una de las técnicas más antiguas y básicas. Por lo general, está unido a una rueda o cilindro del motor para medir la rotación real y estimar la distancia recorrida por el objetivo.

- La localización a través de cámaras, [18] se está usando para ciertas aplicaciones específicas actualmente. La cámara debe proporcionar la calidad de imagen suficiente en un entorno con una iluminación decente para un procesamiento efectivo de la imagen. Se debe identificar un objeto de referencia en cada escena para estimar la posición absoluta instantánea del objetivo lejos de ella. Para ello es necesario procesamiento de imágenes y algoritmos computacionales de alto rendimiento.

- La localización basada en el reconocimiento de puntos de referencia o landmarks, [19], es otra técnica eficaz.

- La tecnología de unidades de medición inercial (IMU), [20], es otro concepto que se ha desarrollado y mejorado mucho en las últimas décadas. Cada unidad contiene un acelerómetro de 3 ejes, un giroscopio de 3 ejes y otros sensores. Utiliza la propiedad inercial de una masa para calcular su movimiento. Este concepto está subdesarrollado debido a los errores acumulativos ilimitados de los sensores.

Cada una de las tecnologías previamente mencionadas tiene diferentes ventajas y limitaciones frente al resto. Por lo tanto, si fusionamos dos tecnologías que tienen ventajas de rendimiento complementarias, la precisión de la localización aumentará y los errores de estimación disminuirán. El odómetro visual, por ejemplo, combina los datos de un odómetro y una cámara para mejorar la precisión de localización del objetivo. Este algoritmo de fusión es el que se está aplicando actualmente en el Mars Exploration Rover de NASA. El móvil utiliza dos pares de cámaras estéreo además de odómetros en las ruedas.

Una vez nombrados los distintos sensores usados para localización de robots móviles en interiores, se pasa a hacer una revisión más exhaustiva de los distintos algoritmos utilizados en los últimos años dentro de esta disciplina.

1.2.2.2 Algoritmos de Localización.

Los algoritmos de localización más exitosos para estimar las coordenadas del mapa y la posición del robot involucran técnicas probabilísticas. Estos esquemas implementan un enfoque SLAM (Localización y mapeo simultáneos), donde los datos son recolectados desde una posición desconocida e introducidos en un mapa mientras se usa ese mapa incierto para ayudar en la localización. El trabajo inicial de Smith y Cheeseman [21] introdujo un marco probabilístico para la construcción de mapas y localización. Una de las técnicas de SLAM actuales utiliza un algoritmo de maximización de expectativas (EM) para construir el mapa y localizar el robot. Este algoritmo se puede utilizar para centrarse en la determinación robusta de correspondencias de características. Otro enfoque común de SLAM utiliza los filtros de Kalman para estimar la posición del robot y construir el mapa.

Existen tres desafíos principales en SLAM:

1. `Compensación del ruido del sensor`: sin ruido en el sensor, la navegación por estimación (Dead reckoning), como la odometría, sería un método de localización suficiente. Desafortunadamente, los pequeños errores en el cálculo se acumulan y hacen que el error de posición aumente con el tiempo. Aunque los sensores externos pueden ayudar a limitar este error, estos sensores también proporcionan un error adicional, que debe ser tratado.

2. `Precisión en la asociación de datos`: para los métodos que localizan al robot utilizando información de sensores externos, es necesario establecer correspondencias entre los datos recolectados en diferentes posiciones del robot. Si no pueden determinarse correspondencias precisas, la información del sensor es inútil para la localización. Esto es esencial para una localización y un mapeo precisos y sólidos. La asociación de datos se hace especialmente difícil en presencia del ruido del sensor descrito anteriormente.

3. `Robustez ante errores no modelados`: en cualquier aplicación del mundo real, habrá eventos o lecturas sensoriales que se salgan fuera de los límites de operación normales. Por ejemplo, el deslizamiento de la rueda debido a un terreno difícil puede causar un error de odometría al integrar la rotación de la rueda. Una puerta cerrada o una mesa movida también puede introducir discrepancias en las medidas externas del sensor, que son mucho más grandes de lo que puede explicarse por el ruido del sensor. Es fundamental que los métodos de localización sean robustos y se recuperen de estos tipos de errores sin modelar.

Ahora veamos más en detalle los métodos probabilísticos usados en localización, [29], [30]. En los métodos probabilísticos se busca estimar el estado del robot en el momento actual, k, conociendo el estado inicial y todas las medidas anteriores. Este problema de estimación es una instancia del problema de filtrado bayesiano, en el que se busca construir la densidad de posición posterior al estado actual. En el enfoque bayesiano, esta función de densidad de probabilidad se toma para representar todo el conocimiento que se posee del estado x_k, y de este se puede estimar la posición actual. En resumen, para localizar el robot hay que computar recursivamente la densidad de posición en cada instante de muestreo. Esto se realiza en dos fases:

- **Fase de predicción**. En esta primera fase se usa un modelo de movimiento para predecir la posición actual del robot $p(x_k|Z^{k-1})$, teniendo en cuenta solo el movimiento. Se asume que el estado actual solo de pende del estado anterior (Markov [22]) y de una entrada de control conocida, y que el modelo de movimiento es especificado como una densidad condicional $p(x_k|x_{k-1}, u_{k-1})$. La densidad predicha sobre x_k se obtiene por integración:

$$p(x_k|Z^{k-1}) = \int p(x_k|x_{k-1}, u_{k-1})p(x_k|Z^{k-1})dx_{k-1} \tag{1.6}$$

- **Fase de actualización**: en la segunda fase se usa un modelo de medida para incorporar información de los sensores. Se asume que la medida es condicionalmente independiente de las medidas anteriores Z^{k-1} y que el modelo de medida se da en términos de probabilidad de que el robot se encuentre en cierta posición si se observa esa medida, $p(z_k, x_k)$. La densidad posterior sobre x_k se obtiene usando el teorema de Bayes:

$$p(x_k|Z^{k-1}) = \frac{p(z_k, x_k)p(x_k|Z^{k-1})}{p(z_k, z_{k-1})} \tag{1.7}$$

Tras esta fase de actualización, el proceso se repite recursivamente, hasta obtener una solución. Dependiendo de cómo se decida representar la densidad $p(x_k|Z_(k-1))$, se obtienen algoritmos variados con grandes diferencias:

- **`Filtro de Kalman`**: si el modelo de movimiento y medición se pueden describir usando una densidad gaussiana, y el estado inicial también se especifica como una gaussiana, entonces la densidad p siempre será una gaussiana. En este caso, las ecuaciones anteriores se pueden evaluar de forma cerrada obteniendo el clásico filtro de Kalman [23]. Las técnicas de filtrado de Kalman han probado ser robustas y precisas para mantener el seguimiento de la posición del robot. Debido a su concisa representación (la media y la matriz de covarianza son suficientes para ello) se trata de un algoritmo particularmente eficiente. Sin embargo, no es capaz de manejar correctamente modelos no lineales o no gaussianos, no puede recuperarse ante fallos y no puede tratar con densidades multimodales. Estas dificultades se pueden arreglar usando extensiones no óptimas del filtro de Kalman, pero todas ellas provienen de la suposición restrictiva de densidad gaussiana inherente al filtro de Kalman.

- **`Localización topológica de Markov`**: para superar estas desventajas, diferentes enfoques utilizan esquemas cada vez más ricos para representar incertidumbres. Estos métodos se pueden distinguir por el tipo de desratización usada para representar el espacio de Estados. La localización de Markov [23] [24] se usa para la navegación de pasillos basada en puntos de referencia, donde el espacio de estados se organiza según la estructura topológica del entorno.

- **`Localización de Markov basada en cuadricula`**: para tratar densidades multimodales y no gaussianas con una buena resolución, se puede llevar a cabo una integración numérica sobre una cuadricula de puntos. Esto implica discretizar la parte interesante del espacio de estado, y usarlo como base para una aproximación de la densidad. Estos métodos son potentes, pero sufren de una gran carga computacional y de un compromiso de conocer a priori el tamaño del espacio de estados. Además, la resolución y por tanto la precisión, tiene que ser fijada de antemano. Debido a los requisitos computacionales, no todas las medidas pueden ser procesadas en tiempo real y mucha información relevante sobre el estado es descartadas. Trabajos recientes han conseguido solución a algunos de estos problemas usando octetos para obtener una representación de resolución variable del espacio de estado, [25].

- **`Localización de Monte Carlo`**: finalmente, se puede representar la densidad por un conjunto de muestras que se extraen al azar. Esta es la representación que usa el algoritmo de Monte Carlo [31], en el cual se toman diferentes enfoques para representar incertidumbre. En lugar de describir la función de densidad de probabilidad en sí misma, la representa manteniendo un conjunto de ejemplos que se extraen de ella aleatoriamente. Para actualizar esta representación de densidad con el tiempo, se utilizan los métodos de Monte Carlo inventados en los años setenta, que han sido redescubiertos de forma independiente en el seguimiento de objetivos, estadística y visión computacional.

Los enfoques probabilísticos están entre los candidatos más prometedores que brindan una solución integral y en tiempo real al problema de la localización de robots, pero los métodos actuales aún enfrentan obstáculos considerables. Las técnicas basadas en el filtro de Kalman han demostrado ser robustas y precisas para realizar un seguimiento de la posición del robot. Sin embargo, como se mencionó anteriormente, un filtro de Kalman no puede representar ambigüedades y carece de la capacidad de relocalizar globalmente el robot en caso de fallo. Aunque el filtro de Kalman puede ser modificado de varias maneras para hacer frente a algunas de estas dificultades, los enfoques más recientes han utilizado mejores esquemas para representar la incertidumbre, alejándose de la suposición restringida de densidad gaussiana inherente al filtro de Kalman. El enfoque de localización de Markov basado en cuadricula puede representar densidades de probabilidad arbitrariamente complejas a una buena resolución. Sin embargo, la carga computacional y los requisitos de memoria son considerables. Además de la desventaja de que el tamaño de la cuadricula, y por tanto, también la precisión a la que puede representar el estado, debe ser fijo de antemano.

El método de Localización de Monte Carlo es el método seguido en este trabajo, ya que, usando una representación basada en muestras, se obtiene un método de localización que tiene varias ventajas frente a otros métodos previos:

- En contraste a las técnicas basadas en el filtrado de Kalman, es capaz de representar distribuciones multimodales y por lo tanto puede localizar globalmente un robot.
- Reduce drásticamente la cantidad de memoria requerida comparada con la localización basada en cuadrícula de Markov, y puede integrar medidas a una frecuencia considerablemente mayor.
- Es más preciso que la localización de Markov con un tamaño de celda fijo, ya que el estado representado en las muestras no está discretizado.
- Fácil implementación.

1.2.3 Métodos de Planificación Local

Los métodos de planificación local se utilizan para evitar colisiones entre el robot y los objetos del entorno. Esta evitación de obstáculos suele basarse en mapas locales. La necesidad de dotar a los robots móviles con la capacidad de evadir obstáculos no conocidos surge del hecho de que la mayoría de los entornos en el mundo real son variables. Es muy común que existan objetos en un entorno que no aparecen en el mapa correspondiente a este, o que algunos de los objetos que sí aparecen en él hayan sufrido modificaciones. Esto da lugar a que algoritmos de planificación de trayectoria basados en la información proporcionada por el mapa sean incapaces de conducir al robot a su destino de manera fiable. En consecuencia, a lo largo de las ultimas décadas el interés por algoritmos que resuelvan este tipo de situación ha ido en aumento, dando lugar a la aparición de diversos métodos de planificación de trayectoria local.

A continuación se hace una revisión a los algoritmos de evitación de obstáculos desarrollados hasta la fecha por orden cronológico:

- **`Algoritmos BUG`**: Los algoritmos de la familia Bug [32] son los más conocidos para resolver el problema de la navegación en un entorno de dos dimensiones desconocido. Llevan al robot de un punto A a un punto B ambos conocidos, sin ninguna información del entorno, o se paran en caso de que el objetivo sea inalcanzable, realizando una serie de simplificaciones, tales como tomar objetos como puntos y los sensores como ideales libres de ruido. No está pensado para su uso en un entorno con obstáculos en movimiento, motivo por el que no se ha implementado este algoritmo.

  -`BUG1`: se recorre el obstáculo alrededor por completo una vez y luego se deja el obstáculo en el punto más cercano al objetivo.

  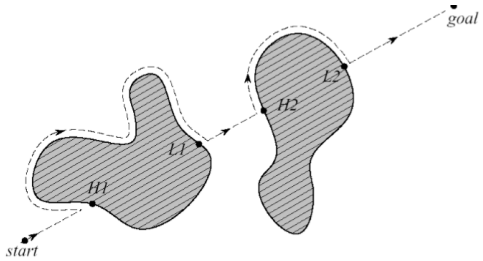

  Figura 1.9: Representación algoritmo Bug1.

  -`BUG2`: se recorre el obstáculo alrededor siempre en el mismo sentido, y se deja cada vez que se cruza la recta que une el objetivo con el punto de partida.

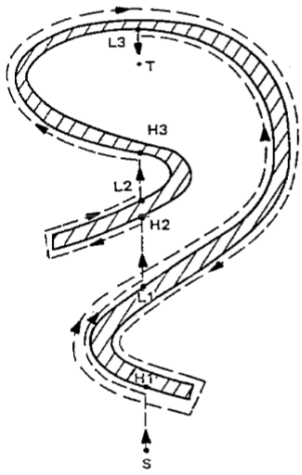

Figura 1.10: Representación algoritmo Bug2.

- **`Elastic Bands`**: [33], se crea un camino deformable y libre de colisiones mediante burbujas. Primero se conecta el inicio y destino en el espacio libre (1.11, a). A continuación se aplica una fuerza de estiramiento en la banda para reducir el camino (1.11, b) y otra de repulsión generada por los obstáculos (1.11, c). Estas fuerzas se ejecutan en tiempo real en presencia de objetos móviles para generar caminos cotos y con giros suaves (1.11, d). La clave de este algoritmo reside en la burbuja, que impone una distancia mínima a los obstáculos. Se impone la restricción de que cada burbuja se tiene que solapar con otras dos (una delante en el camino planificado y otra detrás). El robot puede moverse dentro de cada burbuja sin riesgo de colisión, lo que permite determinar el radio de cada burbuja. Cuando un objeto móvil interfiere en el camino, se deforma añadiendo burbujas.

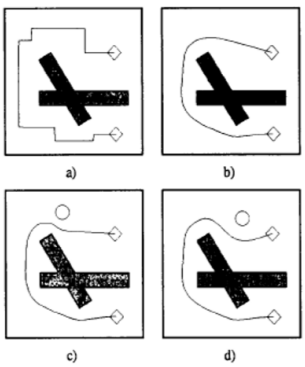

Figura 1.11: Representación algoritmo Elastic Bands.

- **`Vector Field Histogram (VFH)`**: [34], [35]. El entorno local del robot síncrono se representa en una rejilla con 2 grados de libertad, cuyas celdas almacenan la probabilidad de contener un obstáculo. Esta se reduce a un histograma polar en 1 grado de libertad. Los pasos a seguir serían los siguientes: se calcula dicho histograma en todas las direcciones de giro, se buscan todos los pasos disponibles y, por último, se selecciona el que menor función de coste G tiene. Esta función de coste tiene la siguiente forma:

G = a·dirección_objetivo+b·orientación_ruedas+c·dirección_previa

Donde: - *Dirección_objetivo* es la alineación del robot con respecto al objetivo.

- *Orientación_ruedas* es la diferencia entre la nueva dirección y la orientación actual de las ruedas.

- *Dirección_previa* es la diferencia entre la dirección previamente seleccionada y la nueva dirección.

- *a,b,c* son los parámetros de ajuste de comportamiento del robot.

Para el cálculo del histograma polar, se lleva a cabo los siguientes pasos:

- El valor de la certidumbre de cada celda activa c_{ij} se trata como un vector obstáculo con una dirección β_{ij} y una magnitud m_{ij}.

- Se construye el histograma obteniendo el sector que contiene cada celda activa y el valor del histograma se obtiene de sumar las magnitudes de los vectores obstáculos en cada sector.

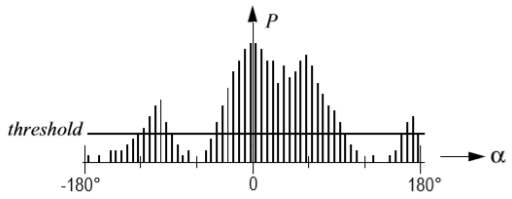

Figura 1.12: Histograma polar.

El algoritmo VFH+ se trata de una simplificación de la dinámica del robot, considerando que el robot solo se mueve en rectas o arcos, y que los obstáculos que bloquean una dirección dada también bloquean todas las posibles trayectorias a través de esa dirección.

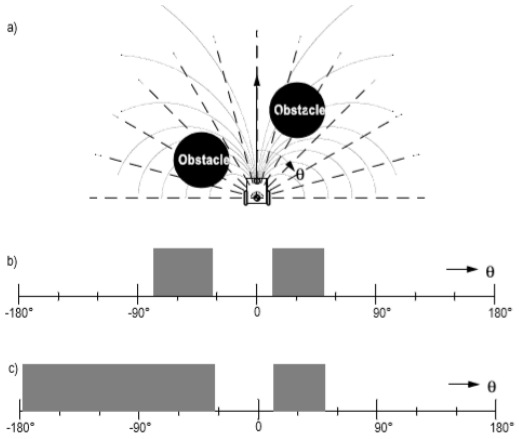

Figura 1.13: Ejemplo de direcciones bloqueadas y histogramas polares resultantes. (a) Robot y obstáculos. (b) Histograma polar (c) Histograma polar enmascarado.

- **Curvature Velocity Method (CVM)**: [36]. Se trabaja con robots síncronos (t_v, r_v) en un espacio transformado de velocidad C. Se supone que el robot navega describiendo arcos de curvatura c $= t_v/r_v$ y se imponen restricciones físicas al robot en cuanto a aceleración y velocidad máximas, impidiendo que el robot vaya hacia atrás (tv>0). Se define una función objetivo para seleccionar la velocidad óptima:

  f(tv,rv) = a1·speed(tv) + a2·dist(tv,rv) + a3·head(rv)

  Cada uno de los términos se normaliza entre 0 y 1. El primer término prima las velocidades de traslación altas hacia el destino, el segundo busca recorrer grandes distancias sin impactar con obstáculos cercanos, y el tercero intenta orientar al robot hacia la dirección del destino. Los términos 'a' son moduladores del comportamiento.

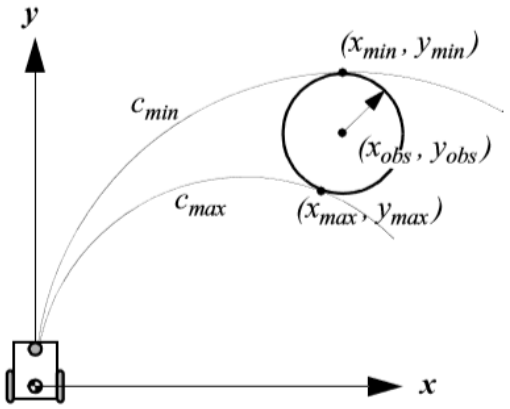

Figura 1.14: Curvas tangentes para un obstáculo.

  Para el cálculo de la distancia mínima primero se convierten los obstáculos al espacio de velocidades (dv=dc si tv $\neq$ 0). Si hay más de uno se obtiene una función de distancia acumulada (D(tv,tv,obs) = min$_o$*bs* $\in$OBSdv(tv,tv,OBS)). Se aplica una distancia límite L (3m) para evitar el cálculo de infinitos valores. Por último, con el fin de simplificar esta función, se convierten los obstáculos en círculos.

  Sin embargo, este método presenta diversos problemas. Por un lado, describir arcos constantemente hace que sea poco eficiente, la navegación con entornos basados en pasillos estrechos presenta muchos fallos, ya que en ocasiones el robot se dará la vuelta y no entrará en el pasillo. Además, no es eficiente en tiempo de cómputo. Por ello, se hace una simplificación de cálculos de distancia, asignando distancias fijas a intervalos que se solapan. Se toma la distancia mínima suponiendo que la curvatura está entre *Cmin* y *Cmax* y para robustecer ante el ruido se aumentan los obstáculos un margen de seguridad.

- **Lane Curvature Method (LCM)**: [37]. Presenta algunas mejoras frente al CVM. Por un lado, se mejora el método al considerar que el robot no navega únicamente en arcos. Los pasillos o carriles se calculan evaluando la longitud y anchura del carril al objeto más cercano y el que tenga mejores propiedades se elige usando una función objetivo. Aunque sigue teniendo el problema de mínimos locales como el CVM, se mejora el funcionamiento en la navegación con pasos estrechos.

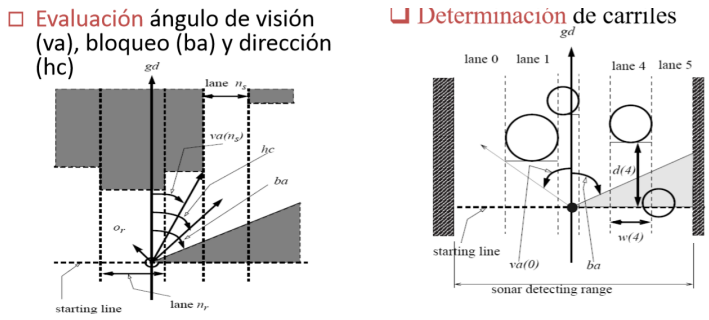

Figura 1.15: Fase de evaluación y de determinación de carriles en algoritmo LCM.

- **Algoritmo de ventana dinámica (DWA)**: [38]. Usa un espacio transformado de velocidad (v,w) y con la cinemática del robot determina el tamaño de la ventana donde seleccionar la velocidad óptima a través de una función objetivo: O = a·heading(v,w)+b·velocity(v,w)+c·dist(v,w) Este algoritmo también asume que el robot se mueve en arcos y se asegura de que este se detiene antes de golpear ningún obstáculo.

- **Aproximación de ventana dinámica global**: [39]. Hasta ahora todos los algoritmos tienen problemas a la hora de conseguir el objetivo global, ya que se basan en sensores locales. Para solventar este problema se añade un planificador global denominado NF1 a la función objetivo O presentada en el DWA. La rejilla de ocupación se actualiza en base a la medida de los sensores locales de distancia (láser y ultrasonidos):

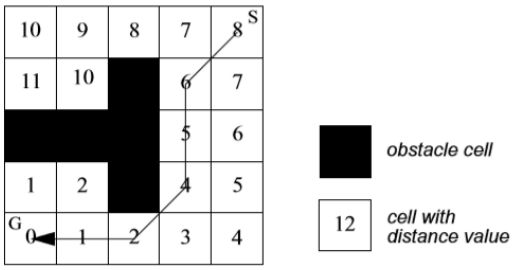

Figura 1.16: Función NF1. S, start; G, goal.

- **Aproximación Schlegel**: [40]. Es otra variación del DWA que tienen en cuenta la forma del robot. Sigue considerando el movimiento del robot en forma de arco e incluye el planificador global NF1, pero este funciona en tiempo real mediante tablas pre-calculadas.

- **Nearness Diagram (ND)**: [41]. Diseñado para entornos densos, altamente poblados y complejos con robot síncrono. Se basa en la estrategia de *divide y vencerás* para reducir la dificultad de la tarea y emplea el paradigma *situated-activity*, el cual resuelve el proceso percepción-acción. No presenta el problema de la coordinación de acciones de tiempo real, ya que las acciones resuelven el problema de la situación y son completas. Se define un conjunto de criterios, situaciones y sus correspondientes acciones (HSGR, HSWR, HSNR, LS1, LS2):

  - Criterio1: seguridad (*Low Safety,High Safety*)

  - Criterio 2: distribución del obstáculo peligroso

– Low Safety 1 (LS1): el obstáculo en la zona de seguridad está solo en un lado del hueco del área de paso ->alejar el robot del obstáculo más cercano pero por el área libre hacia destino.

– Low Safety 2 (LS2): el obstáculo en la zona de seguridad está en ambos lados del hueco del área de paso ->centrar robot entre obstáculos más cercanos hacia destino.

- Criterio 3: destino en el área de paso

– High Safety Goal in Region (HSGR): el destino está en el área de paso libre ->mover el robot hacia el destino.

- Criterio 4: ancho del área de paso

– High Safety Wide Region (HSWR): el destino está en un área de paso ancha ->mover el robot a lo largo del obstáculo.

– High Safety Narrow Region (HSNR): el destino está en un área de paso estrecha ->mover el robot por la parte central del área de paso.

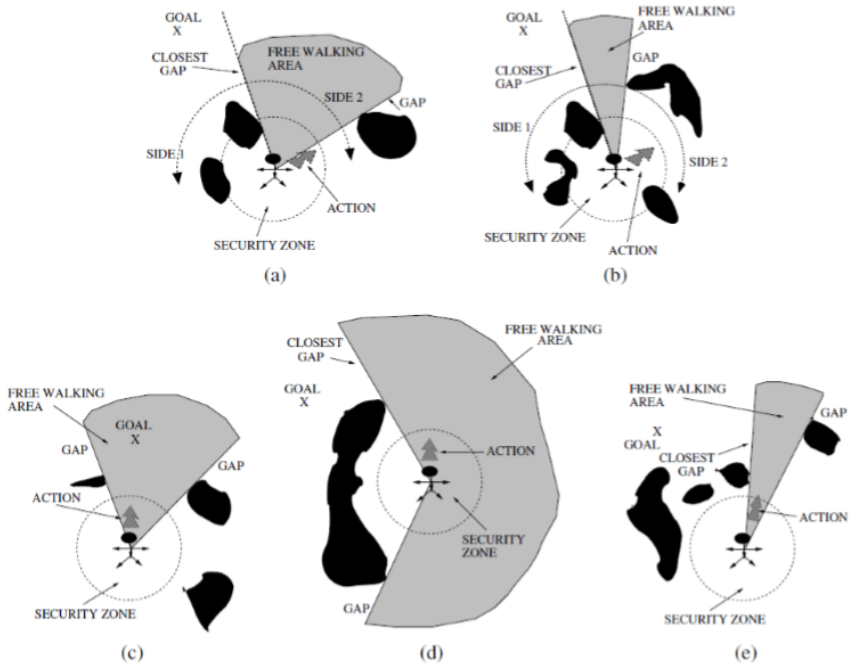

Figura 1.17: (a)LS1, (b)LS2, (c)HSGR, (d)HSWR, (e)HSNR

1.2.4 Métodos de Planificación Global

El concepto de planificación global se define como el proceso para construir o planificar la ruta que lleve al robot a cada una de las submetas determinadas por el control de misión, según las especificaciones del problema a resolver. Esta planificación es una aproximación al camino final que se va a seguir, ya que en la realización de esta acción no se consideran los detalles del entorno local al vehículo. Mientras que la planificación local se lleva a cabo en tiempo de ejecución, la construcción de la ruta global puede realizarse antes de que el vehículo comience a ejecutar la tarea.

A continuación se revisan los distintos métodos de planificación, los cuales se pueden dividir en algoritmos clásicos o determinísticos y algoritmos probabilísticos. Todos ellos se fundamentan en una primera fase de representación del entorno con algún tipo de grafo (identificar un conjunto de rutas dentro del espacio libre), celdas (discriminar entre espacio libre y ocupado) o un campo potencial (imponer una función matemática sobre el espacio disponible) a partir de un mapa, topológico, métrico o mezcla de ambos, lo suficientemente bueno para realizar la navegación. Posteriormente se emplea un algoritmo de búsqueda que encuentra el camino óptimo según cierta función de coste.

En cuanto a los métodos clásicos o determinísticos encontramos los siguientes algoritmos:

- **`Grafo de visibilidad`**: [42]. Se trazan segmentos desde todos los vértices de los obstáculos hasta los vértices que son visibles, incluyendo los puntos inicial y final, eligiendo como ruta optima el camino de menor longitud. Para evitar colisiones con los obstáculos, se aumenta el tamaño de estos de forma virtual antes de realizar la planificación.

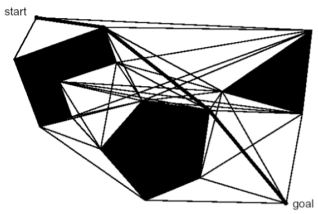

Figura 1.18: Grafo de Visibilidad.

- **`Diagrama de Voronoi`**: [43]. Se tiende a maximizar la distancia entre el robot y los obstáculos del espacio de configuración. Se calcula la distancia desde todos los puntos en el espacio libre hasta los obstáculos y se seleccionan los de mayor distancia, eligiendo como ruta optima el camino de longitud menor.

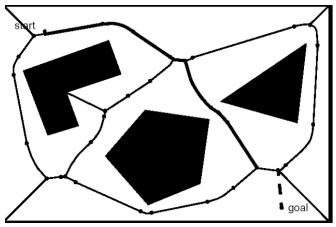

Figura 1.19: Diagrama de Voronoi.

- **Descomposición en celdas (grafo de conectividad)**: [44]. Se precisa la resolución de dos problemas: la descomposición del espacio libre en celdas y la construcción de un grafo de conectividad. El primero de ellos implica construir unas celdas con determinada forma geométrica tal que resulte fácil de calcular un camino entre dos configuraciones distintas pertenecientes a la celda, y la comprobación para averiguar si dos celdas son adyacentes debe disfrutar de la mayor simpleza posible. Aparte de estas características, la descomposición global del espacio libre implica que no deben existir solapamientos entre celdas y que la unión de todas ellas corresponde exactamente al espacio libre. El grafo de conectividad es un grafo no dirigido, y su construcción está asociada a la descomposición en celdas efectuada en el paso anterior, de tal forma, que los nodos van a ser cada una de las celdas, existiendo un arco entre dos celdas si y solo si son adyacentes.

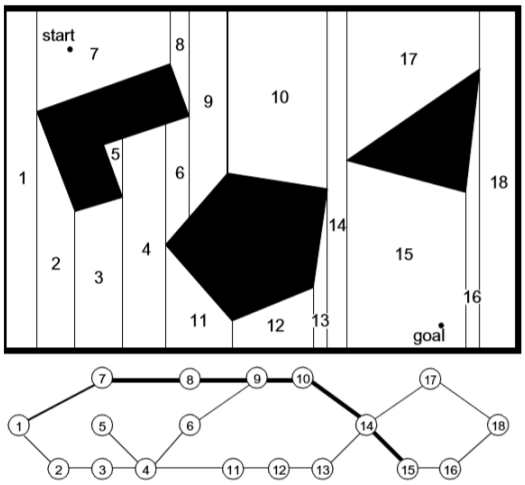

Figura 1.20: Descomposición en celdas.

Una vez especificado el grafo de conectividad, sólo queda emplear un algoritmo de búsqueda en grafos, para la detección de la celda que contiene la posición a la cual se desea llegar, tomando como partida la que contiene la posición inicial. Los distintos métodos basados en este principio, se distinguen por la forma en la cual realizan la descomposición en celdas y como se construye el grafo de conectividad. El método más sencillo de descomposición del espacio libre del entorno en celdas resulta el denominado descomposición trapezoidal.

- **Campos potenciales**: [46]. Los métodos basados en campos potenciales poseen una concepción totalmente distinta a los expuestos más arriba al estar basados en técnicas reactivas de navegación.

  La teoría de campos potenciales considera al robot como una partícula bajo la influencia de un campo potencial artificial, cuyas variaciones modelan el espacio libre. La función potencial en un punto del espacio euclídeo, se define sobre el espacio libre y consiste en la composición de un potencial atractivo, que atrae al robot hacia la posición destino, y otro repulsivo que lo hace alejarse de los obstáculos. La fuerza artificial a la que afecta el vehículo en la posición p, por el potencial artificial U(p) resulta: F(p)=- δ U(p). Al igual que la función potencial, la fuerza artificial es el resultado de la suma de una fuerza

de atracción, proveniente de la posición destino, y otra fuerza de repulsión debidas a los obstáculos del entorno de trabajo.

Así, la navegación basada en campos potenciales se basa en llevar a cabo la siguiente secuencia de acciones:

- Calcular el potencial que actúa sobre el vehículo en la posición actual según la información recabada de los sensores.

- Determinar el vector fuerza artificial.

- En virtud del vector calculado construir las consignas adecuadas para los actuadores del vehículo.

La iteración continua del ciclo expuesto proporciona una navegación reactiva basada en campos potenciales. El problema en este tipo de métodos deviene en la aparición de mínimos locales, es decir lugares que no son la posición destino en los cuales el potencial resulta nulo. Una situación de este tipo puede hacer que el robot quede atrapado en una posición que no sea la destino, o bien debido a la naturaleza discreta del método girar alrededor de ella. Solucionar este conflicto implica definir ciertas funciones potenciales que eviten la aparición de mínimos locales, lo cual resulta arduo, si bien existen soluciones que lo aseguran en entornos donde los obstáculos están modelados mediante círculos.

Existe un método extendido de campo de potencial. En este se introduce un campo de potencial de rotación, en el que la fuerza es función de la orientación del robot al obstáculo; y un campo de potencial de tarea, en el que se filtran los obstáculos que no influyen en el movimiento del robot, incluyendo únicamente los que están dentro de un sector Z delante del robot.

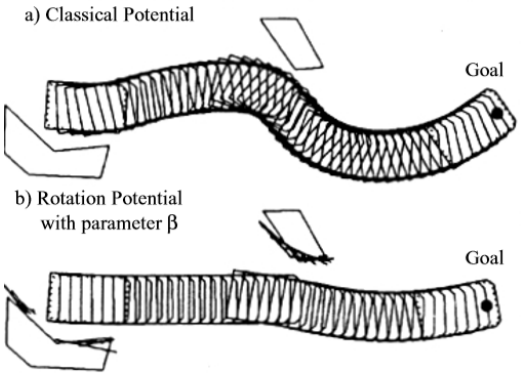

Figura 1.21: Comparación entre campo potencial clásico y extendido.

Por otro lado, los algoritmos probabilísticos se basan en la Teoría de Decisiones. Esta se deriva de dos problemas fundamentales: incluso conociendo el estado actual, hay incertidumbre sobre el resultado de las acciones y, además, normalmente no se conoce el estado actual, sino una estimación. Por tanto se plantea un problema de planificación bajo condiciones de incertidumbre que afectan tanto al resultado de las acciones como a la percepción del estado.

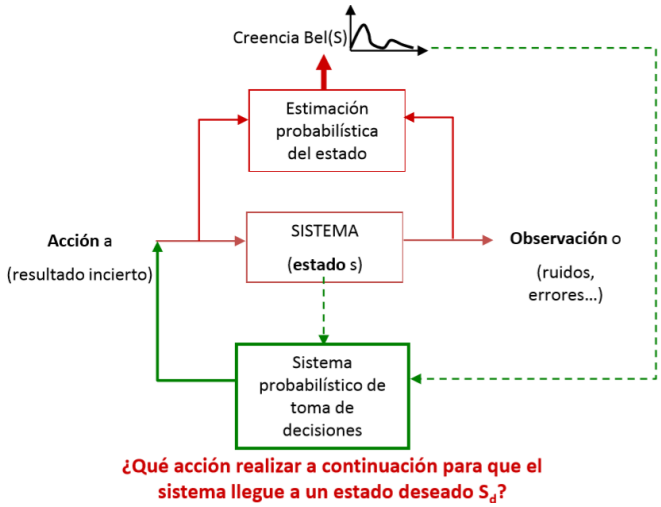

Figura 1.22: Teoría de Decisiones.

De este problema se derivan dos métodos probabilísticos: Proceso de Decisión de Markov (MDPs) [47] [48] y Proceso de Decisión de Markov parcialmente observable (POMDP) [49] [50] [51] [52].

Se define un sistema (estado s) y un agente (decisión), de manera que mediante el estado s se toma la decisión de realizar la acción a, que lleva al sistema a un nuevo estado s. Se debe cumplir la propiedad de Markov:

$P(s_{t+1}|s_0, a_0, s_1, a_1, ..., s_t, a_t) = p(s_{t+1}|a_{t+1})$

Normalmente existe incertidumbre respecto a los resultados de una decisión o acción, esta incertidumbre se modela mediante el modelo de transición como una probabilidad de llegar al estado s' dado que se encuentra en s y se realiza la acción a: T (p(s'|s,a)). Así mismo, definimos el modelo de recompensa R(r(s,a)) como la utilidad de las acciones para conseguir el objetivo.

Normalmente el robot puede sensar el ambiente para observar en qué estado se encuentra (modelo de los sensores). Existen dos casos principales: si se observa directamente el estado, usamos el MDP; si, por el contrario, se tiene una incertidumbre sobre el estado, usamos el POMDP.

Dado el modelo de transición y el modelo de los sensores, el objetivo es encontrar la política óptima a=π^*(s) para maximizar la recompensa futura. Una política es una función que asigna una acción a cada estado. Se considera que las probabilidades de transición sólo dependen del estado actual por lo que son procesos markovianos.

Para encontrar la política óptima en un ambiente observable (MDP) existen varios algoritmos, entre los que cabe destacar el Algoritmo de Iteración de Valores. La idea básica consiste en calcular la recompensa de cada posible estado y usar éstas para seleccionar la acción óptima en cada estado. La recompensa de un estado depende de la secuencia de acciones tomadas a partir de dicho estado de acuerdo a la política establecida. Otros métodos de solución son *iteración de política* y *programación lineal* (al transformar el problema a un problema de optimización lineal).

Sin embargo, en muchos problemas reales, no se puede observar exactamente el estado del sistema, por lo que se tiene un POMDP. Además de los elementos de un MDP, un POMDP incluye: un modelo de observación que especifica la probabilidad de las observaciones dado

el estado, P(O|S); y una distribución de probabilidad inicial para los estados, P(S). El enfoque exacto para resolver un POMDP requiere considerar toda la historia de observaciones y acciones. Esto es equivalente a considerar la distribución de probabilidad sobre los estados y en base a esta determinar las decisiones óptimas. Para ello, se puede considerar un POMDP como un MDP en que los estados corresponden a la distribución de probabilidad. El problema es que el espacio de estados se vuelve infinito y la solución exacta es muy compleja. Las soluciones adoptadas en robótica (métodos heurísticos aproximados) son: planificación por suposición, métodos basados en MDPs o aprendizaje por refuerzo.

1.2.5 Entornos de programación para robots

Una vez revisado el Estado del Arte en navegación de robots móviles, se procede a nombrar y explicar brevemente algunas de las plataformas software más conocidas y utilizadas para programar y controlar el movimiento de un robot. Frameworks como Robot Operating System (ROS) [6] [7], Advanced Robot Interface for Applications (ARIA) [8] y Microsoft Robotics Development Sudio (MRDS) [9] ofrecen a los usuarios las herramientas para el desarrollo de aplicaciones software para robots. Así mismo, la Robotics System Toolbox de Matlab puede usarse como entorno de programación al conectarse con ROS, de manera que permite usar sus propias funciones y algoritmos para facilitar esta tarea.

A continuación, se resumen las características principales de cada uno de ellos:

- **`Robot Operating System`**: Se trata de un framework de código abierto que actualmente es ampliamente utilizado por universidades y empresas de todo el mundo. Se basa en un funcionamiento distribuido y modular mediante nodos, que se comunican entre sí suscribiéndose o publicando en topics. Permite la escritura de aplicaciones en C, C++ y Python. ROS se puede definir de igual manera como un metasistema operativo, ya que es un OS que se instala sobre otro. Lo más usual es que sea un sistema UNIX (Ubuntu, Debian).Por tanto, provee las funcionalidades de un OS (Sistema Operativo) en un cluster (muchas computadoras conectadas entre sí).

- **`Advanced Robot Interface for Applications`**: Es una interfaz de programación de aplicaciones de control de robots orientada a objetos para robots móviles inteligentes de la empresa MobileRobots (y ActivMedia). Escrito en C++, ARIA es un software para el acceso fácil y de alto rendimiento al robot y todos sus sensores y efectores. Además incluye muchas utilidades útiles para programación general de robots y una programación multiplataforma (Linux y Windows). Se puede ejecutar en subprocesos múltiples o simples, utilizando su propio envoltorio alrededor de hilos de Linux o de Win32. Se puede acceder a ARIA a diferentes niveles, desde el simple envío de comandos al robot a través de ArRobot hasta el desarrollo de un comportamiento inteligente de alto nivel mediante Acciones. También se incluye una librería auxiliar llamada ArNetworking, la cual proporciona un marco extensible y fácil de usar para la comunicación con programas remotos a través de una red.

- **`Microsoft Robotics Development Studio`**: Desarrollado por Microsoft, presenta una gran novedad frente al resto, la posibilidad de desarrollar aplicaciones sin escribir código. Es un entorno basado en Windows para el control de robots y su simulación. Está dirigido a desarrolladores académicos, aficionados y comerciales, y maneja una gran variedad de hardware de robots. Requiere el sistema operativo Microsoft Windows 7. Está basado en CCR (Tiempo de ejecución de concurrencia y coordinación): una implementación de librería concurrente basada en .NET para administrar tareas paralelas asíncronas. Esta técnica implica el uso de paso de mensajes y un tiempo de ejecución orientado a servicios ligeros, DSS (Servicios de Software Descentralizados), que permite la adaptación y coordinación de múltiples servicios para lograr comportamientos complejos. Sus características incluyen: una herramienta de programación visual, el lenguaje de programación Microsoft Visual para crear y depurar aplicaciones robóticas, interfaces basadas en web y ventanas, simulación 3D (incluida aceleración hardware) y un fácil acceso a sensores y actuadores. El lenguaje de programación principal es C#.

El meta-sistema operativo ROS es el entorno utilizado en el desarrollo de este trabajo junto con la Robotics System Toolbox de Matlab, por tanto, se ha dedicado un apartado a cada uno de ellos en los que se explicarán sus características principales.

El hecho de haber elegido ROS como entorno principal para el proyecto deriva de que, gracias a su carácter modular y distribuido, permite trabajar fácilmente con la *Robotics System Toolbox* de MATLAB. Además, se trata de la plataforma más relevante actualmente en el campo de la robótica. Investigadores, empresas y amantes de la temática, utilizan este software para programar sus robots y, como ha sido mencionado, al ser una plataforma de código abierto, ROS busca crear una comunidad que comparta sus conocimientos adquiridos. Esto tiene grandes ventaja, ya que cuenta con una infinidad de información para cualquier tipo de aplicación que no deja de aumentar y renovarse día a día.

1.3 Objetivos del proyecto

El propósito general del trabajo consiste en realizar un conjunto de aplicaciones relacionadas con la navegación autónoma en un entorno Matlab-ROS. Estas tendrán aplicación en docencia e investigación dentro del Departamento de Electrónica.

Para ello se plantean los siguientes objetivos específicos:

- **Estudio del entorno de desarrollo**: Estudio del entorno de desarrollo robótico ROS y de la Robotics System Toolbox de Matlab.
- **Arquitectura del sistema**: Configuración del entorno de trabajo y comunicación entre las distintas máquinas.
  - **Simulador**: Análisis y creación de un entorno de simulación en STDR de ROS para simular los distintos algoritmos antes de aplicarlos a las plataformas robóticas reales. Para ello será necesario la generación de un mapa del entorno en el que se probará

posteriormente: área oeste, planta 2 del Edificio Politécnico, y pasillo central del mismo.

- **Interfaz con el robot real**: Control básico de los robots reales mediante el envío de comandos de velocidad lineal y angular para poder recorrer el entorno y obtener la lectura de los datos de los sensores del robot.

- **Desarrollo e implementación de los diferentes algoritmos de navegación**: Una vez analizados y controlados el entorno de simulación y las plataformas robóticas, se procede a desarrollar e implementar en ellos los diferentes algoritmos de navegación incluidos en el presente trabajo:

  - **Mapeado del entorno**: Implementación y comparación tres algoritmos de mapeado: mapeado puro asumiendo posición actual conocida, algoritmo de SLAM incluido en la Robotics System Toolbox de Matlab y nodo SLAM_gmapping de ROS.
  - **Localización**: Implementación del Método de Monte Carlo (MCL).
  - **Planificación Local**: Implementación del algoritmo VFH. Creación de una aplicación de seguimiento de pasillos mediante lógica borrosa, usando la Transformada de Hough para detección de líneas. El objetivo de esta es que el robot detecte las líneas de un pasillo con irregularidades y sea capaz de moverse en una trayectoria paralela a estas y centrada.
  - **Planificación global**: Implementación del algoritmo PRM.

- **Pruebas finales**: Una vez funcionando los diferentes algoritmos individualmente, se llevan a cabo diferentes pruebas en las que se realiza una aplicación completa donde el robot recorre el mapa evitando obstáculos (VFH) hasta que se localiza en este (MCL), y crea una ruta hasta el destino (PRM) al que llega mediante un controlador *Pure Pursuit*.

Además, una vez conseguidos los objetivos iniciales, se realiza una ampliación del proyecto utilizando un simulador muy novedoso, especialmente pensado para conducción autónoma, CARLA. Este incorpora un *ros-bridge* que permite trabajar con ROS. Se propone así, como objetivo adicional, la creación de una aplicación de control de trayectoria basado en el seguimiento de waypoints pre-establecidos que será presentada al primer concurso publicado por el equipo creador del simulador, el CARLA Challenge. Para ello, se establecen los siguientes objetivos específicos:

- **Generación de trayectoria**: Creación de una trayectoria a partir de waypoints precomputarizados mediante una interpolación de tipo spline.
- **Control para el seguimiento de trayectoria**: Diseño del controlador.
- **Percepción de la escena y comportamiento reactivo**: Procesamiento de los datos recibidos del láser y las cámaras para la detección de semáforos y objetos con el fin de dotar al vehículo de un comportamiento reactivo.

- **Creación del agente en Python para la participación en el CARLA Challenge**: Adaptación de la aplicación anterior a la estructura requerida en el concurso. Para ello es necesario crear un agente en Python y encapsular toda la aplicación en una imagen docker que será enviada al Challenge. Este lanza todos los nodos, lleva a cabo todas las inicializaciones, recoge y publica en topics los datos de los sensores y envía los comandos de aceleración y dirección al vehículo simulado en CARLA.

En la siguiente sección se detalla el contenido y estructura del trabajo.

1.4 Estructura de la memoria

El presente documento está dividido en varios capítulos y secciones, de los cuales esta introducción se corresponde con el primero, en el que se presenta el contexto, haciendo un estudio del estado del arte en el área de navegación autónoma, motivación y objetivos del proyecto.

En el segundo capítulo se desarrollan las herramientas utilizadas, tanto las plataformas robóticas Pioneer como el entorno de desarrollo: ROS y Robotics System Toolbox de Matlab. Como ampliación se hace un análisis del novedoso simulador CARLA.

En el tercer capítulo se describe en profundidad cómo se ha desarrollado este trabajo. Se comienza explicando la arquitectura del sistema y se procede desarrollando uno a uno cada algoritmo implementado.

Las conclusiones del proyecto están recogidas en el capítulo 4, así como algunas líneas futuras que puedan derivarse de este trabajo.

Como anexos se encuentran el manual de usuario, los planos, el pliego de condiciones y el presupuesto necesario para la elaboración del trabajo.

El manual de usuario explica los pasos necesarios para poder reproducir el trabajo realizado, de manera que pueda entenderse mejor.

En la sección de planos se introducen los esquemáticos de los láseres incorporados a las dos plataformas robóticas, así como la estructura de directorios de los algoritmos más significativos.

El pliego de condiciones contiene los requisitos tanto de hardware como de software necesarios para la realización del proyecto.

Por último, en el apartado de presupuesto se estima el coste para la realización del trabajo.

Capítulo 2

Herramientas utilizadas

Sólo la propia y personal experiencia hace al hombre sabio.

Sigmund Freud

En este capítulo se abordará un estudio de las herramientas utilizadas a lo largo del proyecto. Primero se hablará de las dos plataformas robóticas utilizadas para implementar y comparar la robustez de los algoritmos desarrollados: Amigobot y Seekur. Seguidamente se explican los dos entornos de programación utilizados para la elaboración de estos: ROS y la *Robotic System Toolbox* de MATLAB, así como la conectividad entre estos.
Por último, se introduce como ampliación una descripción del novedoso simulador CARLA.

2.1 Robots Pioneer

En esta sección se describen los dos robots reales utilizados para implementar los algoritmos realizados primeramente en simulación. Se tratan de robots Pioneer, plataformas populares para educación, investigación, creación de prototipos, exhibiciones y otros proyectos creados por la empresa *MobileRobots* [62] . Concretamente, los dos robots usados son el Amigobot y el Seekur.

2.1.1 Plataforma Amigobot

El AmigoBot es un robot basado en una arquitectura de estilo ARCS muy simplificado con el objetivo de que sea asequible para aplicaciones multi-robot y clases de robótica. Posee seis sonares anteriores y dos posteriores para telemetría y encoders para seguir la posición del robot (x,y,θ). Su diseño de tracción diferencial proporciona una buena movilidad.

El AmigoBot incorpora un microcontrolador Hitachi H8S con sistema operativo ARCOS. Este es el encargado de procesar la información del sónar y la odometría, así como de gestionar la comunicación con una Raspberry PI incorporada a través del puerto serie. En esta Raspberry PI se instaló ROS y en el arranque lanza principalmente el máster de ROS (roscore), el driver *p2os* para gestionar la comunicación con el Amigobot, y el driver *rplidar* para la lectura de los

datos del láser. La Raspberry PI se conecta al RPLIDAR mediante USB y presenta una IP fija para conectarse a ella desde PC's externos vía WiFi.

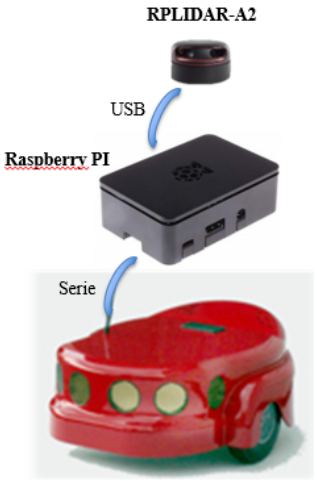

Figura 2.1: Arquitectura software del Amigobot y sensores incorporados.

La carga útil máxima es de 1kg. El AmigoBot ha sido diseñado como un sistema cerrado para disuadir la manipulación interna, al contrario que otras plataformas creadas con visión de extensibilidad. Incluye librerías ARIA API para C++ bajo sistema operativo Linux o WINDOWS2000/XP.

Entre las aplicaciones principales para las que se ha pensado su uso se encuentran: aplicaciones en enseñanza para cursos de introducción a robótica e investigación.

Para ejecutar cada AmigoBot con un sistema serie Ethernet se requiere un ordenador de mínimo 300 MHz con un puerto serie libre, sistema operativo WIN 2000/XP o Linux, un disco duro con al menos 11Mb de espacio libre, lector de CD ROM y un punto de acceso a Ethernet IEEE 80211b.

La plataforma es prácticamente holonómica y puede rotar en el lugar moviendo ambas ruedas, o puede mover las ruedas solo en un lado para formar un circulo de 27 cm de radio. Puede operar marcha atrás.

Un AmigoBot cargado completamente puede ejecutarse hasta 2h, siendo el tiempo de recarga de 6h. En el microcontrolador incluido en el AmigoBot se tiene un bus de entrada/salida de 8-bits para hasta 16 dispositivos. Hay dos puertos series, uno de los cuales está disponible para un accesorio adicional y el otro es usado para comunicarse con el microprocesador externo.

Figura 2.2: Plataforma Amigobot.

2.1.1.1 Láser RP-LIDAR A2

Al AmigoBot usado en este trabajo, el cual puede verse en la figura 2.2, se le incorporó un láser RP-LIDAR A2. El RPLIDAR A2 adopta un sistema de medición por triangulación láser desarrollado por SLAMTEC, y por lo tanto tiene un excelente rendimiento en todo tipo de ambientes interiores y exteriores sin exposición directa a la luz solar.

Puede tomar hasta 4000 muestras de láser por segundo con alta velocidad de rotación. Y equipado con la tecnología patentada OPTMAG de SLAMTEC, trabaja de manera estable durante mucho tiempo.

El sistema puede realizar escaneos 2D de 360 grados dentro de un rango de 6 metros. Los datos de nubes de puntos 2D generados se pueden usar en mapeo, localización y modelado de objetos y entornos.

La frecuencia de escaneo típica del RPLIDAR A2 360 ° Laser Scanner es 10hz (600rpm). En esta condición, la resolución será de 0.9 °. Y la frecuencia de escaneo real se puede ajustar libremente dentro del rango de 5-15hz según los requisitos de los usuarios. El RPLIDAR A2 consta de un núcleo de escáner de rango y partes de alimentación mecánica que hacen que el núcleo gire a gran velocidad. Cuando funciona normalmente, el escáner girará y escaneará en el sentido de las agujas del reloj. Los usuarios pueden obtener los datos de escaneo de alcance a través de la interfaz de comunicación del RPLIDAR y controlar el inicio, la detención y la velocidad de rotación del motor de rotación a través de PWM.

El RPLIDAR A2 viene con un sistema de detección de velocidad de rotación y adaptativo. El sistema ajustará la resolución angular automáticamente de acuerdo con la velocidad de rotación real.

En el anexo *Planos*, se incluye la sección C.1, donde se incluyen los esquemáticos del láser RP-LIDAR A2.

2.1.2 Plataforma Seekur

Seekur es una plataforma robusta, para todo tipo de condiciones meteorológicas que puede manejarse desde en campos abiertos hasta en garajes. Su forma única y su dirección omnidireccional permite un movimiento verdaderamente holonómico, lo que significa que puede girar en su propia longitud o incluso salir hacia un lado si se encuentra bloqueado por delante y por detrás. Ideal para navegación inteligente, el Seekur ofrece espacio, potencia y conexión en red hasta para cinco PCs de formato EBX. Esto abre camino para procesamiento de imagen a bordo, comunicaciones basadas en radio, telemetría láser, DGPS, y otras funciones autónomas. Puede ejecutarse hasta siete horas gracias a sus baterías de níquel-cadmio. Sus cuatro ruedas montadas en una suspensión de hierro han sido diseñadas para velocidades de hasta 2.2m/s y cuestas de hasta el 20 %, cargando incluso 50kg extra.

El paquete base de Seekur incluye:

- Seekur base: suspensión de acero resistente, construcción de aluminio de una sola pieza (ahorro de peso), dirección omnidireccional en 4 ruedas, tracción en las 4 ruedas, interfaz de usuario LCD con pantallas de diagnóstico y retroiluminación, conversión DC-DC de alta potencia para accesorios.

- Servidor de control de PC

- Seekur OS de bajo nivel

- Software de cliente API de nivel medio ARIA

- Batería duradera de 24V de níquel-cadmio

- Cargador

- Manual de operaciones

Además Seekur ofrece una amplio gama de opciones:

- Mapeado y navegación láser

- Unidad de medida inercial

- Hasta cinco PC a bordo

- Radiocomunicaciones inalámbricas

- Joystick inalámbrico

- DGPS

El cuerpo de aluminio de bajo peso coloca los componentes de la suspensión de acero en la parte baja del robot. Esto reduce el centro de masa para un mejor manejo del terreno y capacidad de nivelación. Los neumáticos de 400mm de diámetro se manejan en interiores, exteriores y en medio de ambos.

Seekur puede subir cuestas del 20 % y pasos de 100 mm. En superficies niveladas, Seekur puede moverse a velocidades de 2.2m/s, con cargas útiles de hasta 70 kg. Las cargas útiles incluyen todos los accesorios y se deben equilibrar adecuadamente para que este o cualquier robot funcione

de manera efectiva. Puede operar aproximadamente 7 horas con una batería completamente cargada. El tiempo de recarga es de 8 horas con el cargador de velocidad estándar.

El firmware que se ejecuta en un microcontrolador endurecido para el medio ambiente controla la plataforma. Acepta comandos y transfiere el estado y las lecturas del sensor a un microprocesador incorporado a bordo del robot que hace de cliente utilizando el mismo protocolo Aria que todas las demás plataformas MobileRobots.

Figura 2.3: Plataforma Seekur

2.1.2.1 Láser SICK LMS151

Los escáneres láser LMS1xx 2D forman parte de la amplia gama de tecnología de medición láser de SICK y ofrecen una alternativa eficiente y económica frente a otras soluciones en aplicaciones tanto interiores como exteriores. SICK ofrece estos sensores de medición láser en un gran número de variantes. Los escáneres LMS1xx hacen frente con facilidad a las diferentes exigencias de alcance y software de aplicación. Asimismo, las variantes de seguridad completan perfectamente la gran gama de aplicaciones. Todas las variantes de esta gama de productos se caracterizan por su diseño y dimensiones compactas, así como por su bajo peso. Su tamaño compacto integra una excelente innovación: la tecnología Multi-Echo aumenta la disponibilidad de los sensores en el exterior y permite montarlos incluso detrás de un cristal.

En concreto, la plataforma Seekur cuenta con un láser LMS151, el más completo dentro de esta gama. Sus características principales son:

- Divisor de 905nm

- Ángulo de abertura de 270°

- Frecuencia de exploración de hasta 50Hz

- Resolución angular de 0.25°

- Rendimiento: 0.5 m hasta 50 m

- Tiempo de respuesta: mayor o igual a 20ms

- Alcance: 18m con 10 % de reflectancia y 50m con 90 % de reflectancia

- Número de ecos: 2

- Error sistemático de ±30mm, aleatorio de 12mm

Figura 2.4: Láser SICK LMS151.

En el anexo *Planos*, se incluye la sección C.1, donde se incluyen los esquemáticos del láser SICK LMS151.

2.2 Robot Operating System (ROS)

Una vez descritas las plataformas sobre las que se implementarán los algoritmos desarrollados, se procede a documentar el entorno de programación base para la mayoría de aplicaciones robóticas, ROS, utilizado en este trabajo como método de unión entre la plataforma final y el código desarrollado con la *Robotics System Toolbox* de Matlab. Así mismo, se describen las herramientas más importantes incorporadas por ROS que han sido utilizadas durante la elaboración del proyecto.

En la figura 2.5 se puede observar un esquema de en qué consiste la programación clásica de aplicaciones robóticas.

2.2.0.1 Generalidades

Robot Operating System o ROS es una plataforma de desarrollo robótico que permite crear aplicaciones con múltiples sensores y actuadores de forma flexible. Es una colección de herramientas, bibliotecas y convenciones que tiene como objetivo simplificar la tarea de crear un comportamiento de robot complejo y robusto en una amplia variedad de plataformas robóticas. Este se creó desde cero para fomentar el desarrollo de software de robótica colaborativa.

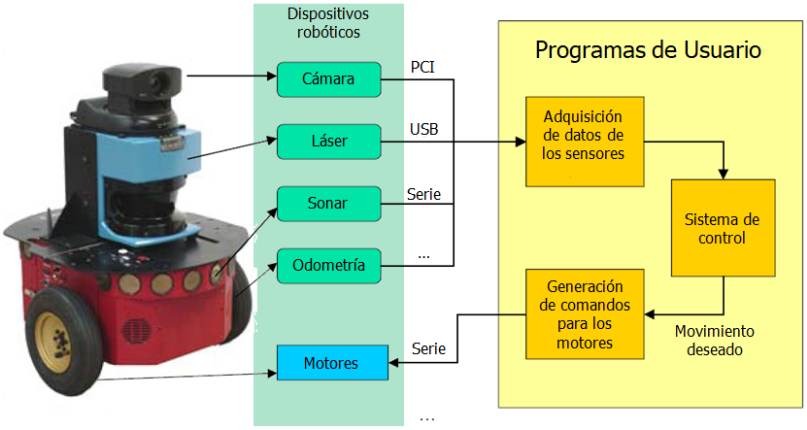

Figura 2.5: Programación de aplicaciones robóticas.

Se organiza en distribuciones (una al año) de manera similar a la distribución de Linux Ubuntu. Las liberadas en años pares son LTS y tienen un soporte de 5 años, mientras que las liberadas en años impares tienen un soporte de 2 años. Cada versión LTS de ROS está ligada a una versión LTS de Ubuntu. En este proyecto se trabaja con la versión LTS ROS Kinetic Kame (2016) funcionando bajo Ubuntu 16.04 LTS.

ROS provee la funcionalidad de un sistema operativo en un cluster heterogéneo. Proporciona los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores. Se desarrolló originalmente en 2007 en el Laboratorio de Inteligencia Artificial de Stanford.

Por tanto, tiene dos partes básicas: la parte del sistema operativo, ros, y ros-pkg que, como se ha indicado anteriormente, se trata de una suite de paquetes aportados por la contribución de usuarios que implementan funcionalidades como localización y mapeo simultáneo, planificación, percepción, simulación, etc.

ROS fue diseñado para ser lo más distribuido y modular posible, de manera que los usuarios puedan usar tanto o tan poco como deseen. Esta modularidad permite escoger qué partes se consideran útiles y cuáles se prefiere implementar por cuenta propia. Su naturaleza distribuida fomenta una gran comunidad de paquetes proporcionados por usuarios que dan un gran valor añadido al sistema ROS. Estos paquetes cubren aplicaciones desde la implementación de nuevos algoritmos hasta drivers y aplicaciones para industria.

Siempre existe un nodo principal de coordinación, *roscore* o *máster*. El resto de nodos están perfectamente distribuidos, permitiendo procesamiento distribuido en múltiples núcleos, multi-procesamiento, GPUs y clusteres. Se trabaja mediante publicación o subscripción de flujos de datos: imágenes, estéreo, láser, control, actuador, contacto, etc.

En 2012 comenzó el proyecto de código abierto ROS-Industrial, que amplía las capacidades avanzadas de ROS a la automatización industrial y la robótica. El repositorio de ROS-industrial incluye interfaces para manipuladores industriales comunes, pinzas, sensores y redes de dispositivos.

A continuación se describen algunos de los conceptos básicos de ROS:

- `ROS Master`: Es el gestor de ROS, por lo que siempre ha de estar presente en el sistema. Permite que los nodos se localicen entre sí y registra los nombres de topics y servicios, entre otras funciones.
- `Nodos`: Son programas ejecutables que facilitan el diseño modular (cada nodo se compila y ejecuta individualmente). Se escriben utilizando una librería cliente de ROS (roscpp en C++ o rospy en Python). Estos pueden suscribirse o publicar en *Topics* y pueden proporcionar o utilizar *Servicios*.
- `Topics`: Son canales de comunicación utilizados por los nodos para comunicarse entre sí. Cada topic transporta un único tipo de mensaje.

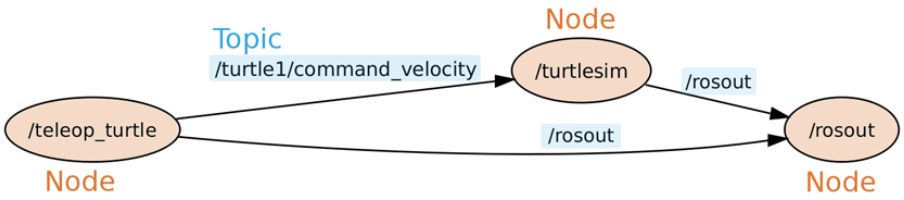

Figura 2.6: Relación entre nodos mediante topics.

- `Mensajes`: Tipo estructurado de datos para la comunicación entre nodos. Existen mensajes ya definidos para los tipos de datos más utilizados (láser, odometría, mapas de ocupación, etc.), además, pueden definirse nuevos mensajes propios de cada aplicación.
- `Servicios`: Modelo de comunicación síncrona entre nodos de tipo cliente/servidor. Se utilizan para eventos especiales o que necesiten confirmación.

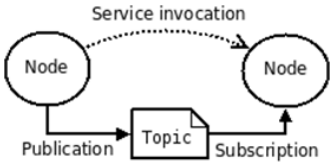

Figura 2.7: Modelo de comunicación entre nodos mediante Topics y Servicios.

- `Rosbag`: Conjunto de herramientas que permiten almacenar y reproducir datos de una ejecución del sistema.
- `TF`: Topic y mensaje especial de ROS que sirve para relacionar los distintos marcos de coordenadas. El marco de coordenadas se relacioina con el topic mediante el campo *frame_id* del mismo.

En el Manual de Usuario, en la sección A.3, se detallan las herramientas y comandos más útiles para trabajar con nodos y topics, así como con las transformadas.

2.2.0.2 Simulador STDR

ROS no está ligado a un simulador específico, si no que incluye tres diferentes: Gazebo, Stage y STDR.

Una de las herramientas clave durante el desarrollo del presente trabajo es el simulador STDR, en el que se realizaron todas las pruebas iniciales, previas a su implementación en las plataformas reales.

STDR, acrónimo de *Simple Two Dimensional Robot Simulator*, permite modelar y simular diferentes robots y sistemas sensoriales dentro de un entorno de movimiento bidimensional. No se trata del simulador más realista o con mayores funcionalidades disponibles actualmente; sin embargo, presenta algunas ventajas: es un simulador sencillo, ligero y de fácil uso, con capacidad multirrobot y multisensor y que posee un entorno gráfico programado en QT. Sus principales limitaciones son la falta de colisiones entre robot, la falta de modelado dinámico y su funcionalidad estrictamente bidimensional.

La intención es crear una simulación lo más simple posible, minimizando las acciones que el investigador ha de realizar para empezar sus experimentos. Además, STDR puede funcionar con o sin un entorno gráfico, lo que permite realizar experimentos incluso usando conexiones ssh. Al ser creado específicamente para ROS, presenta unas características muy útiles a la hora de trabajar con él. Cada robot y sensor emite una transformada de ROS (tf) y todas las medidas son publicadas en topics de ROS. De esta manera, STDR utiliza todas las ventajas de ROS. Además, STDR puede trabajar junto con el visualizador de ROS *Rviz*. En la siguiente figura puede observarse el entorno gráfico de este simulador.

2.2.0.3 Visualizador Rviz

Rviz es un programa que sirve para visualizar los distintos datos que existen dentro de una ejecución de ROS. Este programa está dotado de plugins que permiten representar los tipos de datos más comunes, tales como odometría, láser, sónar, mapas, trayectorias, puntos, etc. Esta herramienta es clave a la hora de depurar un programa ejecutándose en ROS.

Rviz permite al programador ver lo que el robot está viendo, pensando y haciendo. Desarrollar un robot puede ser una tarea muy ardua si no se sabe exactamente lo que el robot cree que está pasando en el entorno. Permite ver el mundo a través de los ojos del robot, ya sean estos cámaras, láseres o encoders. Hay dos maneras de pasar la información a Rviz. Por un lado, Rviz entiende la información de sensores y estado como medidas del láser, nubes de puntos, cámaras y sistemas de coordenadas. Estos tienen diferentes formas de visualización que nos permiten configurar cómo nos gustaría ver plasmada la información.

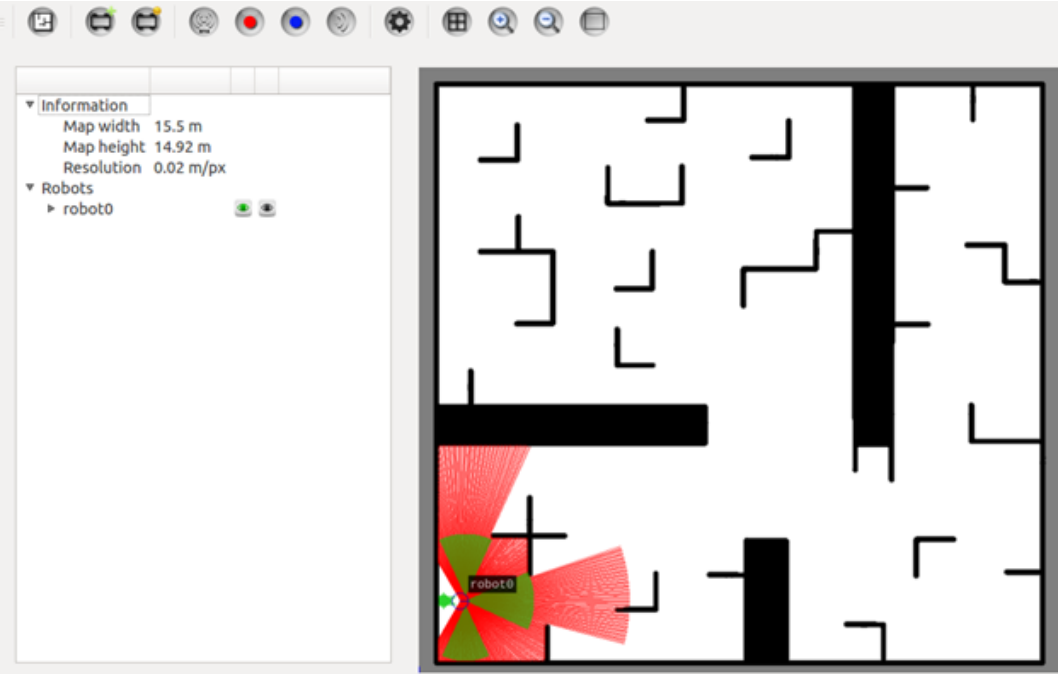

Figura 2.8: GUI del simulador STDR.

Por otro lado, tenemos marcadores de visualización que nos permiten visualizar información proveniente de distintos topics con distintas formas, tamaños y colores (flechas, cubos, círculos, etc.). La combinación de los datos de sensores y los marcadores hacen de Rviz una herramienta muy potente para el desarrollo de capacidades del robot e investigación. Por todo esto, se puede concluir que Rviz es una potente herramienta de código abierto para depurar diferentes aplicaciones de robótica.

A continuación se hace una descripción de la interfaz gráfica de Rviz, representada en la figura 2.9 .

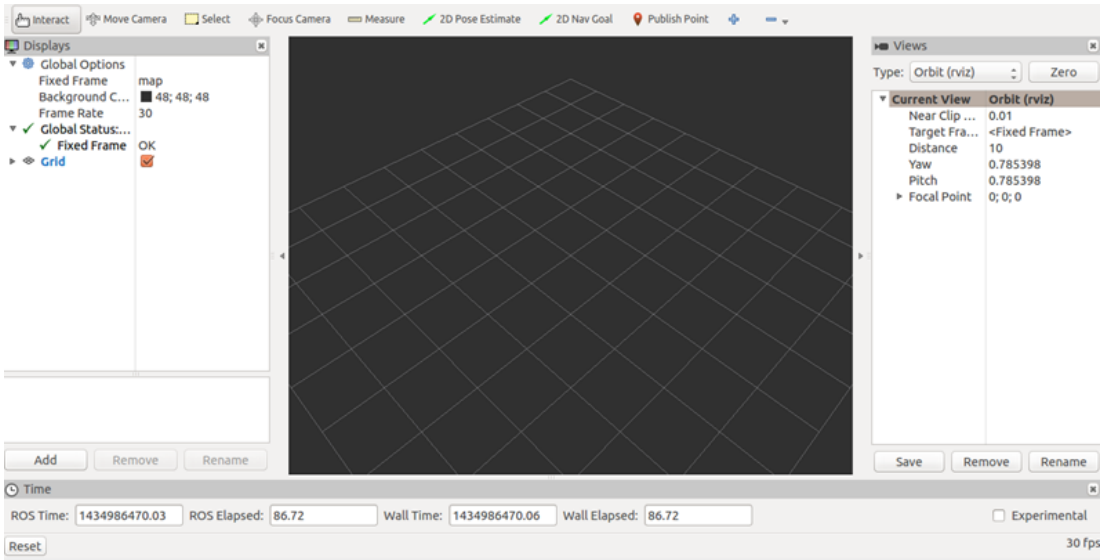

Figura 2.9: Interfaz gráfica de RVIZ

La pantalla puede dividirse en cinco secciones:

- La barra superior nos permite cambiar entre herramientas para mover la visualización, obtener datos de la misma o enviar algunos comandos de posición.

- La barra inferior muestra el tiempo de ROS.

- En la barra izquierda se añaden los distintos datos a representar. Los cuales pueden añadirse por topic o por tipo y en los que se puede modificar su apariencia como su color o tamaño.

- En la barra derecha se puede cambiar el punto de vista.

- En la ventana central se muestra la visualización, que por defecto será una rejilla vacía.

Algunos de los tipos de datos que se pueden visualizar son los siguientes:

- `Map`: representa un mapa de tipo rejilla de ocupación. Existen dos formatos:
  - Map: representa en negro las celdas ocupadas, en gris claro las libres y en gris oscuro las desconocidas.

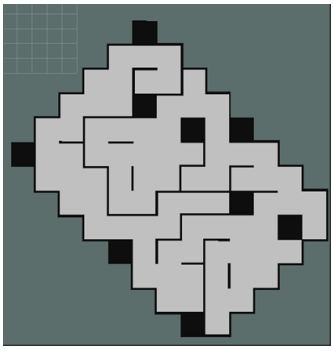

Figura 2.10: Representación Map

  - Costmap: visualiza la ocupación de cada celda en distintos colores

- `Odometría`: representa la posición del robot dentro de un mapa como una flecha. Para obtener el recorrido se puede aumentar el numero de posiciones acumuladas a representar (variable Keep). También se pueden variar el tamaño y color de las flechas para representar distintos robots.

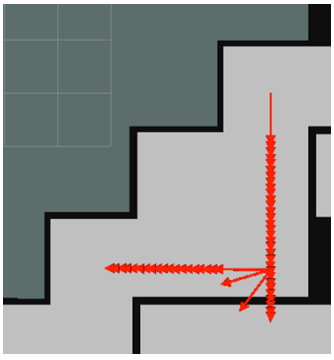

Figura 2.11: Representación Odometría

- Láser: representa las medidas del láser, pintando sobre el mapa los impactos del láser haciendo uso de las transformadas entre marcos de coordenadas. Podemos modificar el tamaño de los impactos y su color.

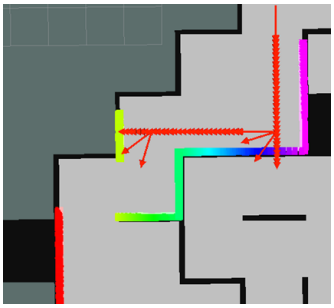

Figura 2.12: Representación Láser

- Sensores de distancia: representa las medidas del sónar o ultrasonidos como un cono dependiendo de su apertura y con una distancia proporcional a su medida.

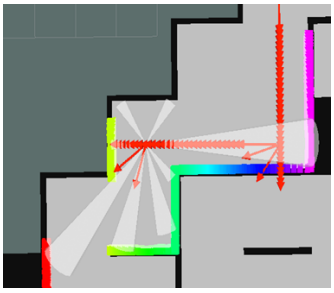

Figura 2.13: Representación Sónar

- Árbol de transformadas (tf): representa el eje de coordenadas de cada uno de los marcos de coordenadas en el sistema así como las relaciones entre ellos.

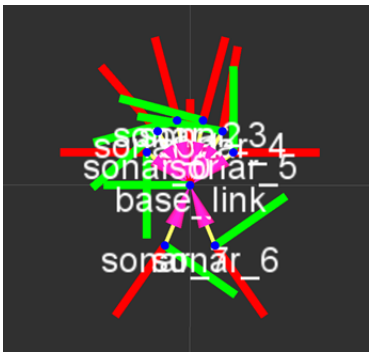

Figura 2.14: Representación tf

2.2.0.4 Paquete P2OS

Para gestionar la comunicación entre el microcontrolador del Amigobot y la Raspberry PI, es necesario utilizar el driver *p2os*.

El paquete P2OS es un conjunto de drivers, modelos y utilidades para robots usando el interfaz P2OS/ARCOS y protocolo Pioneer.

P2os_driver contiene p2os, el nodo principal que gestiona la comunicación con el microcontrolador del Pioneer. Los parámetros que usa son:

- Port

- use_sonar: para decidir si usa sonar o no

- Pulse: controla la frecuencia de comprobación de comandos y deshabilitación de motores.

Topics a los que se subscribe:

- /cmd_vel

- /gripper_control

- /cmd_motor_state

- /ptz_control

Topics en los que publica:

- /sonar - /pose - /gripper_state - /motor_state - /aio

- /ptz_state

- /tf

- /battery_state

- /dio

Con esto, ya hemos visto las herramientas de ROS necesarias para la realización de este trabajo. En la siguiente sección se pasa a explicar la *Robotics System Toolbox* de Matlab así como la conectividad del sistema completo.

2.3 Robotics System Toolbox de Matlab

Robotics System Toolbox proporciona algoritmos y conectividad de hardware para el desarrollo de aplicaciones de robótica para vehículos, manipuladores y robots humanoides.

Las herramientas del sistema proporcionan una interfaz entre MATLAB/Simulink y ROS. De esta manera podremos realizar aplicaciones sencillas de manera mas rápida que programando en ROS nativo. Además, soporta la generación de código C++, permitiendo generar un nodo Ros a partir de un modelo Simulink y desplegarlo automáticamente en una red ROS.

En el siguiente apartado se explica más detenidamente la conectividad entre estos dos sistemas.

2.3.0.1 Conectividad Matlab-ROS

Como se explicó en el apartado 2.2.0.1, ROS presenta una arquitectura de programación distribuida, en la que diferentes nodos se comunican entre sí a través de mensajes enviados a los topics. ROS permite que estos nodos se distribuyan en diferentes máquinas comunicadas por una red, para ajustarse a los recursos disponibles.

Configurar ROS en un sistema multi-máquina es sencillo. Solo es necesario un Máster de ROS (roscore) que se ejecutará en una de las máquinas, siendo esta conocida por el resto de máquinas del sistema. Esto se indica configurando en todas ellas la variable de entorno ROS_MASTER_URI mediante la siguiente sentencia:

```
export ROS_MASTER_URI=http://IP_ROSMASTER_MACHINE:11311
```

donde IP_ROSMASTER_MACHINE es la dirección IP de la máquina que ejecuta el Máster. Los topics creados por cada una de las máquinas serán accesibles a todas las demás, siempre y cuando exista conectividad completa entre ellas a través de la red. Para ello, es necesario que cada máquina comunique su propia dirección de red dentro del sistema ROS a través de la variable de entorno ROS_IP:

```
export ROS_IP=IP_LOCAL_MACHINE
```

donde IP_LOCAL_MACHINE es la dirección IP de la propia máquina.

Estas variables de entorno se configuran en el fichero *.bashrc*.

A modo de prueba se puede ejecutar el comando *rostopic list* en todas las máquinas y comprobar que se tiene acceso a todos los topics creados por todos los nodos desde cualquiera de ellas.

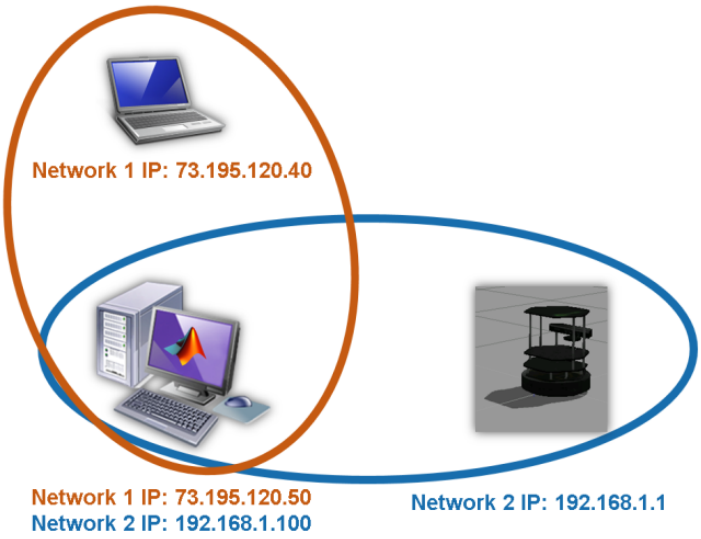

Figura 2.15: Sistema multimáquina.

En este caso particular, trabajaremos con dos máquinas. En una de ellas se ejecutará Matlab y en la otra se ejecutará ROS, bien para trabajar con el simulador o con el robot real. Por tanto, vamos a estudiar concretamente la conectividad Matlab-ROS.

En la máquina en la que se ejecuta ROS, lo primero que se debe hacer es modificar el fichero *.bashrc* para configurar correctamente las variables de entorno con la dirección IP de la máquina, como se explicó anteriormente.

Por otro lado, en el código que programemos en Matlab se debe incluir el siguiente comando para conectar con el sistema ROS:

```
rosinit('http://172.29.30.177:11311','NodeHost','172.29.29.50');
```

Siendo la primera dirección IP la de la máquina ejecutando ROS y la segunda la de la propia máquina. Al acabar el programa hay que desconectarse del sistema ROS mediante la instrucción *rosshutdown*.

Para suscribirnos a un topic hay que hacer uso de la función *rossubscriber('nombre_topic')*, mientras que para publicar se usará *rospublisher('nombre_topic', 'tipo_mensaje')*. El mensaje se crea con *rosmessage('nombre_publisher')* y para publicarlo se utiliza la siguiente función: *send('nombre_Publisher', 'nombre_mensaje')*.

Para leer de los sensores se usará la función *receive('nombre_subscriber')*. También podemos acceder al último dato leído en un Subscriber mediante la instrucción *subscriber.LatestMessage*.

Por otro lado, para definir la periodicidad del timer se utiliza el objeto *r=robotics.Rate(frecuencia_deseada)*. De esta forma, para ejecutar un bucle de forma periódica añadiremos la instrucción *waitfor(r)* dentro del bucle.

A continuación se nombran otras funciones útiles para comunicarse con ROS:

- `Rostopic`: obtiene información sobre topics de ROS
- `Rosmsg`: obtiene información sobre mensajes de ROS y sus tipos
- `readMessages`: lee mensajes de un rosbag
- `select`: selecciona un subconjunto de mensajes de un rosbag
- `transform`: transformación de marcos de coordenadas
- `waitForTransform`: espera hasta que la transformación está disponible
- `getTransform`: obtiene la transformación entre dos sistemas de coordenadas
- `sendTransform`: envía la transformación a la red ROS
- `rostf`: obtiene, envía y aplica transformaciones
- `readImage`: convierte un dato de imagen de ROS en una imagen en Matlab
- `writeImage`: escribe una imagen de Matlab en un mensaje tipo imagen de ROS
- `lidarScan`: crea un objeto para almacenar datos en 2D de un láser
- `plot`: muestra gráficamente los datos de un láser
- `readCartesian`: convierte las lecturas del láser en coordenadas cartesianas

Algunos objetos importantes son:

- `LaserScan`: crea un mensaje de tipo laserScan

- `lidarScan`: crea un objeto para almacenar medidas del láser en 2D

- `OccupancyGrid`: crea un mensaje de mapa de ocupación

En el siguiente apartado se hace un estudio de los algoritmos incluidos en la *Robotics System Toolbox*.

2.3.0.2 Algoritmos para robots móviles

Los algoritmos de *Robotics System Toolbox* de Matlab se centran en aplicaciones robóticas móviles. Estos permiten crear mapas de entornos utilizando rejillas de ocupación, realizar localización y mapeo simultáneos, desarrollar la planificación de rutas para robots en un entorno determinado y sintonizar controladores para seguir un conjunto de puntos. Además, permite realizar evitación de obstáculos, estimación de estado y localización basada en datos de sensores.

A continuación se hace una breve descripción de cada uno de ellos, los cuales serán explicados más en profundidad en el apartado de desarrollo, donde son usados como base en la mayoría de aplicaciones a desarrollar.

1. **SLAM**: dispone de dos ejemplos de implementación de un algoritmo de SLAM tanto offline como online. En los ejemplos se muestra como implementar un algoritmo de localización y mapeado simultáneos usando medidas obtenidas de un lidar en simulación usando optimización de un gráfico de posición. El objetivo es construir un mapa del entorno usando las medidas del láser y recoger la trayectoria del robot al tiempo que este se desplaza por el entorno.

2. **Seguimiento de trayectoria para un robot diferencial**: este ejemplo muestra cómo controlar un robot para que siga una trayectoria deseada. Usa un controlador de tipo Pure Pursuit [70] para conducir un robot simulado a lo largo de un camino predeterminado. Este camino es un conjunto de waypoints definidos explícitamente o calculados usando un planificador de trayectoria. Se crea un controlador Pure Pursuit para un robot simulado de tipo diferencial y calcula los comandos de control para seguir el camino dado. Estos comandos se usan para conducir el robot a lo largo de la trayectoria.

3. **Planificación de trayectoria**: en entornos de diferente complejidad. Este ejemplo muestra como calcular un camino libre de obstáculos entre dos posiciones dado un mapa usando un planificador PRM (Probabilistic Roadmap). Este planificador construye un mapa de carreteras en el espacio libre de un mapa previo usando nodos escogidos aleatoriamente y conectándolos entre sí. Una vez que el mapa de carretera ha sido construido, se elige una trayectoria a seguir. En este ejemplo, el mapa se representa como una red de ocupación. Para obtener los nodos en el espacio libre del mapa, PRM usa una representación de tipo red de ocupación binaria para deducir el espacio libre. A la hora de construir el mapa de carreteras libre de obstáculos no tiene en cuenta las dimensiones del robot. Por tanto, hay que aumentar los bordes del mapa tanto como las dimensiones del robot para evitar colisiones.

4. **Monte Carlo Localization**: este ejemplo muestra como aplicar el algoritmo de Monte Carlo Localization en el robot TurtleBot en el simulador Gazebo de ROS. Se trata de un algoritmo para localizar un robot usando un filtro de partículas. El algoritmo requiere un mapa conocido previamente, en el que estimará la posición y orientación del robot dentro del mapa basándose en su movimiento y sensores. El algoritmo empieza con una creencia inicial de la distribución de probabilidad de la posición del robot, que es representada como partículas distribuidas según dicha creencia. Estas partículas son propagadas siguiendo el modelo de movimiento del robot cada vez que este cambia de posición. Según se reciben nuevas medidas de los sensores, cada partícula evalúa su precisión comprobando la probabilidad de recibir dichas medidas de los sensores estando en esa posición. Tras esto, el algoritmo redistribuye las partículas. Este proceso se repite hasta que todas las partículas convergen en una nube única en torno a la posición real del robot si se consigue localizar correctamente.

   Adaptative Monte Carlo Localization (AMCL) es la variante del MCL. Este ajusta dinámicamente el número de partículas basándose en la distancia *KL* para asegurar que la distribución de partículas converge a la posición real basada en todas las medidas de sensores y movimiento anteriores con alta probabilidad.

5. **Mapeado con posiciones conocidas**. Este ejemplo muestra cómo crear un mapa del entorno usando las lecturas de los sensores, asumiendo conocida la posición del robot al tiempo que se realizan estas medidas. Así mismo, muestra cómo usar funciones de conversión de la *Robotics System Toolbox*, como *quat2eul* que convierte los ángulos dados en forma de cuaternio en ángulos de Euler.

6. **Seguimiento de un móvil usando un filtro de partículas**: el filtro de partículas es un algoritmo de estimación bayesiano recursivo implementado en el objeto *robotics.ParticleFilter*. En el ejemplo de *Monte Carlo Localization* se ve la aplicación de este para el seguimiento de la posición del robot en un mapa conocido. En este ejemplo un coche controlado remotamente es seguido en un entorno exterior. La posición del robot es proporcionada por un GPS, que tiene ruido. También se conocen los comandos de movimiento, aunque el robot no ejecutará estos exactamente debido a las imprecisiones de la mecánica o del modelo. Este ejemplo muestra el uso del filtro para reducir los efectos del ruido en las medidas y conseguir una estimación más precisa de la posición del robot.

2.4 Simulador CARLA

Como se indicó en el capítulo anterior, se decidió ampliar el proyecto mediante la elaboración de un algoritmo de navegación depurado y testeado en un nuevo simulador creado para la investigación en el campo de la conducción autónoma, CARLA [63].

Las tecnologías más avanzadas de conducción autónoma solo rinden bien con un conjunto extremadamente limitado de entornos y condiciones climáticas. Uno de los principales problemas es que resulta difícil entrenar vehículos para hacer frente a todas las situaciones. Y las más desafiantes suelen ser las menos comunes. Hay una gran variedad de circunstancias complicadas

a las que los conductores rara vez se enfrentan: un niño que sale corriendo a la carretera, un vehículo que circula en sentido contrario, un accidente que se produce inmediatamente delante, y muchos más. En cada una de estas circunstancias, un coche autónomo debe ser capaz de tomar la mejor decisión, aunque la probabilidad de enfrentarse a ellas sea mínima. Y eso plantea una pregunta importante: ¿cómo pueden los fabricantes automovilísticos entrenar y probar sus vehículos ante eventos tan infrecuentes?

El equipo compuesto por el investigador de Intel Labs, Alexey Dosovitskiy y varios compañeros suyos del Toyota Research Institute y del Centro de Visión por Computador de Barcelona ha creado un simulador de conducción de código abierto que los fabricantes de coches pueden utilizar para probar las tecnologías de conducción autónoma en situaciones realistas.

El sistema, llamado CARLA (siglas del inglés de: Car Learning to Act, o Coche que Aprende a Actuar, en español), simula una amplia gama de condiciones de conducción y repite situaciones de peligro de manera interminable para ayudar al aprendizaje.

Los simuladores de conducción no son nuevos. Ya existen muchas versiones digitales, modelos de carreras realistas y hasta entornos basados en videojuegos. Varios grupos de conducción autónoma los han usado para probar sus tecnologías. Pero ninguno de ellos ofrece el tipo de retroalimentación que los sistemas de conducción autónoma necesitan para capacitarse de manera efectiva. Ni ofrecen un control significativo sobre las condiciones de conducción ni las acciones de otros agentes. Los simuladores de carreras no suelen incluir tráfico cruzado ni peatones. Y los simuladores de ciudades como *Grand Theft Auto* [64] no permiten controlar el clima, la posición del Sol, el comportamiento de otros vehículos, semáforos, peatones y ciclistas, entre otros elementos. Todos ellos son primordiales a la hora de depurar un algoritmo de control para conducción autónoma, ya que todas estas situaciones son comunes en la vida real.

CARLA ofrece una biblioteca de activos que pueden aplicarse en ciudades bajo diferentes condiciones climáticas y de iluminación. La biblioteca incluye 40 edificios diferentes, 16 modelos de vehículos animados y 50 peatones animados. El equipo los ha utilizado para crear dos ciudades con varios kilómetros de carreteras transitables y luego ha probado tres enfoques diferentes para el entrenamiento de sistemas de conducción autónoma.

Esto puede proporcionar un terreno de pruebas útil y seguro para nuevas ideas. Además, al ser de código abierto y uso gratuito, es una forma de fomentar la investigación y el desarrollo en este sector.

A continuación se listan algunas de sus características más importantes:

1. `Escalabilidad a través de una arquitectura multi-cliente`: múltiples clientes en el mismo o distintos nodos pueden controlar diferentes actores.

2. `API flexible`: CARLA posee una API que permite a los usuarios controlar todos los aspectos relacionados con la simulación, incluyendo generación de trafico, comportamiento de peatones, tiempo, sensores y mucho más.

3. `Sensores`: los usuarios pueden configurar diferentes tipos de sensores incluyendo LIDARs, cámaras múltiples, sensores de profundidad y GPS entre otros.

Figura 2.16: Mapas urbanos del simulador CARLA ante diferentes condiciones meteorológicas.

4. `Simulación rápida para planificación y control`: este modo deshabilita el modo sin rumbo para ofrecer una ejecución rápida de una simulación de comportamiento del trafico y la carretera para los cuales no son necesarios gráficos.

5. `Generación de mapas`: los usuarios pueden crear sus propios mapas fácilmente siguiendo las herramientas estándar de OpenDrive como RoadRunner.

6. `Simulación de escenarios de trafico`: su módulo de *ScenarioRunner* permite a los usuarios definir y ejecutar diferentes situaciones de trafico basadas en comportamientos modulares.

7. `Integración con ROS`: CARLA posee una integración con ROS a través del ROS-bridge.

8. `Códigos base para conducción autónoma`: se provee de bases para conducción autónoma como agentes, incluyendo el agente AutoWare [71] y un agente de aprendizaje de imitación condicional.

Con el objetivo de dar a conocer el nuevo simulador y animar a los distintos grupos de todo el mundo dedicados a la investigación de conducción autónoma a probar la nueva herramienta el equipo ha creado el primer CARLA Challenge [66] [65]. Con él se busca empezar a trabajar en los principales objetivos de CARLA: dar apoyo para el desarrollo, entrenamiento y validación de sistemas de conducción en entornos urbanos. Con este Challenge se busca conseguir una buena habilidad en situaciones de trafico realistas.

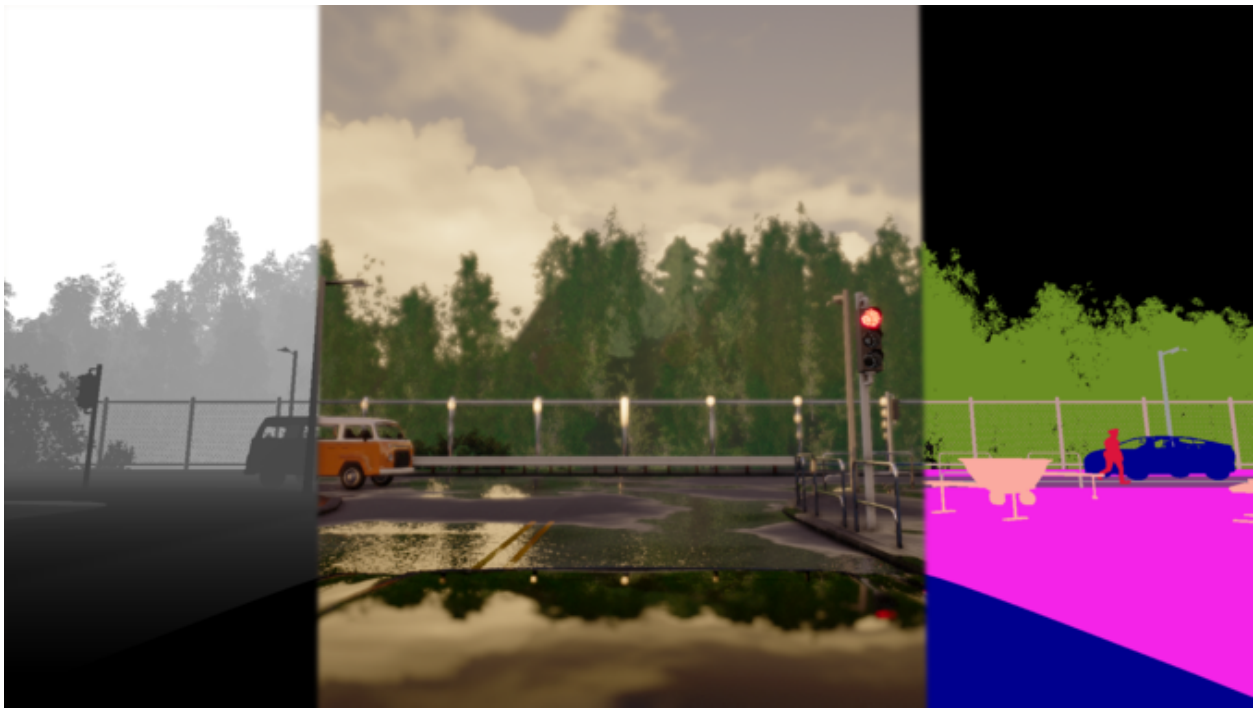

Figura 2.17: El simulador recrea el entorno de un modo realista y permite a los usuarios usar una serie de sensores para el guiado del coche.

Como parte de la ampliación de este trabajo se participó, dentro del grupo Robesafe del Departamento de Electrónica y con la colaboración de sus integrantes, en el Carla Challenge, como se explicará más adelante dentro del capítulo de Desarrollo, en la sección 3.7.

En el siguiente enlace puede verse un vídeo que muestra el entorno gráfico del simulador:

`https://youtu.be/boG9ZR6zPzM`

Una vez explicadas las herramientas utilizadas, así como las aplicaciones y funcionalidades del framework en el que se ha basado el trabajo, se procede a explicar el desarrollo del mismo.

-

Capítulo 3

Desarrollo

A fuerza de construir bien, se llega a buen arquitecto.

Aristóteles

En este apartado se desarrollará y explicará todo el trabajo realizado a lo largo del proyecto. En la primera sección se describe la arquitectura del sistema, tanto en el trabajo en simulación como en los robots reales. Tras esto se dedicará un apartado a cada uno de los algoritmos desarrollados, explicando la base teórica, el framework del que se parte y las comparativas entre los distintos sistemas en los que se implementa así como las ventajas y desventajas entre distintos algoritmos con la misma funcionalidad, en el caso de que se desarrollaran más de uno.

3.1 Arquitectura del Sistema

En esta primera sección, se explica cómo se configura el entorno de trabajo y cómo se realiza la comunicación entre las distintas máquinas mediante topics. Así mismo, se ven las principales diferencias entre el trabajo en simulación y en las plataformas reales.

3.1.1 Trabajo en simulación

En la figura 3.1 se muestra un esquema del sistema distribuido empleado en la realización del trabajo en simulación. En Matlab se desarrollan las aplicaciones, gracias a la sencillez de programación y la no necesidad de compilación. Por otro lado, ROS corre en una máquina virtual en la que se ejecutan el roscore, el simulador y el visualizador. El simulador publica los topics de odometría, láser y sónar, a los que se subscribe la aplicación ejecutándose en Matlab, mientras que esta publica los comandos de velocidad.

El proceso para llevar a cabo dicha conectividad está explicado en el apartado 2.3.0.1.

Figura 3.1: Configuración distribuida en simulación.

El trabajo en simulación se realiza con el simulador STDR explicado en el capítulo anterior, en el apartado 2.2.0.2. Lo primero que se hizo es crear un robot AmigoBot con sus sensores asociados mediante la interfaz gráfica del simulador como se observa en la figura 3.2. Se crea un robot circular con las dimensiones del AmigoBot real, añadiendo un sensor láser y ajustando sus parámetros para simular un RPLIDAR-A (400 haces, rango 0.15m-8m, span 360 grados). Se añadió así mismo un sensor de ultrasonidos (sonar) con sus parámetros (rango 0.1m-5m, span 15 grados, frecuencia 20Hz, posición y orientación) y posteriormente un anillo de ultrasonidos añadiendo otros 7 sonares más modificando el parámetro de orientación y posición. Por otro lado, los mapas en ROS vienen definidos por un archivo imagen y un archivo de descripción .yaml que indica el archivo de imagen, la resolución, el origen del mapa y el umbral para detectar obstáculos y espacios libres según la escala de grises.

Una vez hecho esto se creó el archivo *amigobot.launch*[1], que permite lanzar el simulador STDR con el robot amigobot creado previamente desde la interfaz del simulador. Este incluye otro archivo .launch el cual lanza un nodo que proporciona la descripción del robot a ROS. Crea un nodo que lanza el mapa con el que se trabajará y otro que genera la relación entre los marcos de coordenadas map y world. Finalmente incluye otro archivo .launch que lanza la interfaz gráfica del simulador y crea otro nodo que carga el robot Amigobot en la posición inicial.

A continuación se tienen que especificar los topics a los que nos suscribiremos para leer los sensores, así como los topics en los que se publicará.

Los Subscribers se suscriben a los siguientes topics utilizando la función *rossubscriber(nombre_topic)*:

- Odometría: */robot0/local_odom*
- Láser: */robot0/laser_1*
- Sonar: */robot0/sonar_i* (siendo i de 0 a7 según el sonar al que nos queramos suscribir de los 8 existentes).

[1]En el apartado A.5 dentro del *Manual de Usuario*, se detalla la creación de este fichero.

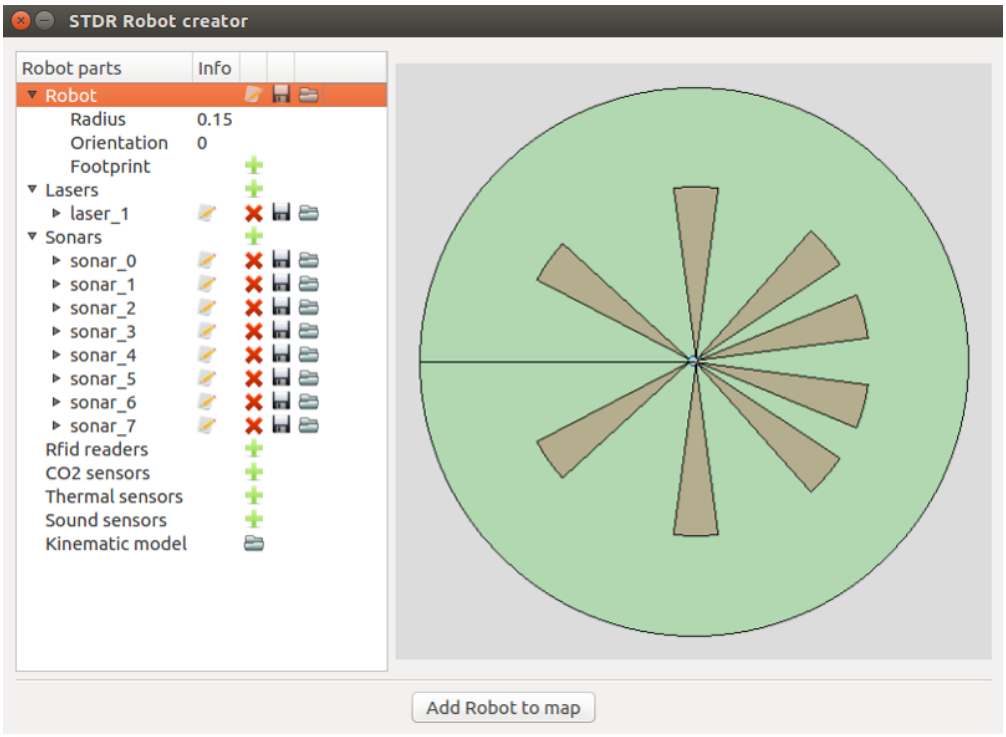

Figura 3.2: GUI para creación del robot.

Para leer la odometría se puede utilizar la instrucción *subscriber.LatestMessage* con la siguiente sintaxis: *odom.LatestMessage.Pose.Pose.Position* para leer la posición, siendo odom el nombre del subscriber suscrito al topic */robot0/local_odom*, o *odom.LatestMessage.Pose.Pose.Orientation* para leer la orientación. Esta orientación vendrá dada en forma de cuaternio, por lo que se tiene que hacer uso de la función *quat2eul* para pasarlo a ángulos de Euler y obtener así el ángulo de interés Yaw.

El único topic en el que se publicará es en el de comandos de velocidad: */robot0/cmd_vel*, cuyo mensaje a publicar es del tipo *geometry_msgs/Twist*.

Se hizo una primera prueba en el script *ini_simulador.m* para asegurarnos de que se recibían mensajes del simulador. Posteriormente se realizó un script de lectura de sensores en *lee_sensores.m*, los cuales se muestran gráficamente mediante el método plot en el caso del láser y por pantalla con *LatestMessage* en el caso del sonar, comprobándose que la lectura es correcta.

Tras esto se creó un script llamado *avanzar.m* que hace avanzar el robot una distancia deseada, con el fin de comprobar que se publicaba correctamente en el topic */robot0/cmd_vel*.

Como última prueba, para comprobar que obteníamos correctamente la orientación del robot transformándolo a ángulos de Euler, se creó un script llamado *girar.m* que hace girar el robot un ángulo especificado. Para ello es necesario crear un bucle que lea la odometría y compruebe el ángulo girado hasta el momento.

Tras estas pruebas, pudimos concluir que la conectividad Matlab-ROS para simulación era correcta.

3.1.2 Trabajo con los robots reales.

Dada la naturaleza de ROS hay que tener en cuenta que el código creado puede ser ejecutado en cualquier plataforma. Por tanto este código puede ser ejecutado en un robot real siempre que se disponga del mismo tipo de sensores. Una vez asegurado el correcto funcionamiento en simulación, volvemos a realizar todas las pruebas con los robots reales.

En la figura 3.3 se muestra un esquema del sistema distribuido empleado en el trabajo con robots reales. En Matlab se desarrollan las aplicaciones, mientras que ROS se ejecuta en el robot real. Así mismo, en caso de ser necesario el uso de ciertas herramientas de ROS como *rqt* o *RVIZ* se puede utilizar la máquina virtual para ejecutar estas. En este caso el robot publica los topics de odometría, láser y sónar, a los que se subscribe la aplicación ejecutándose en Matlab, mientras que esta publica los comandos de velocidad y habilitación de motores.

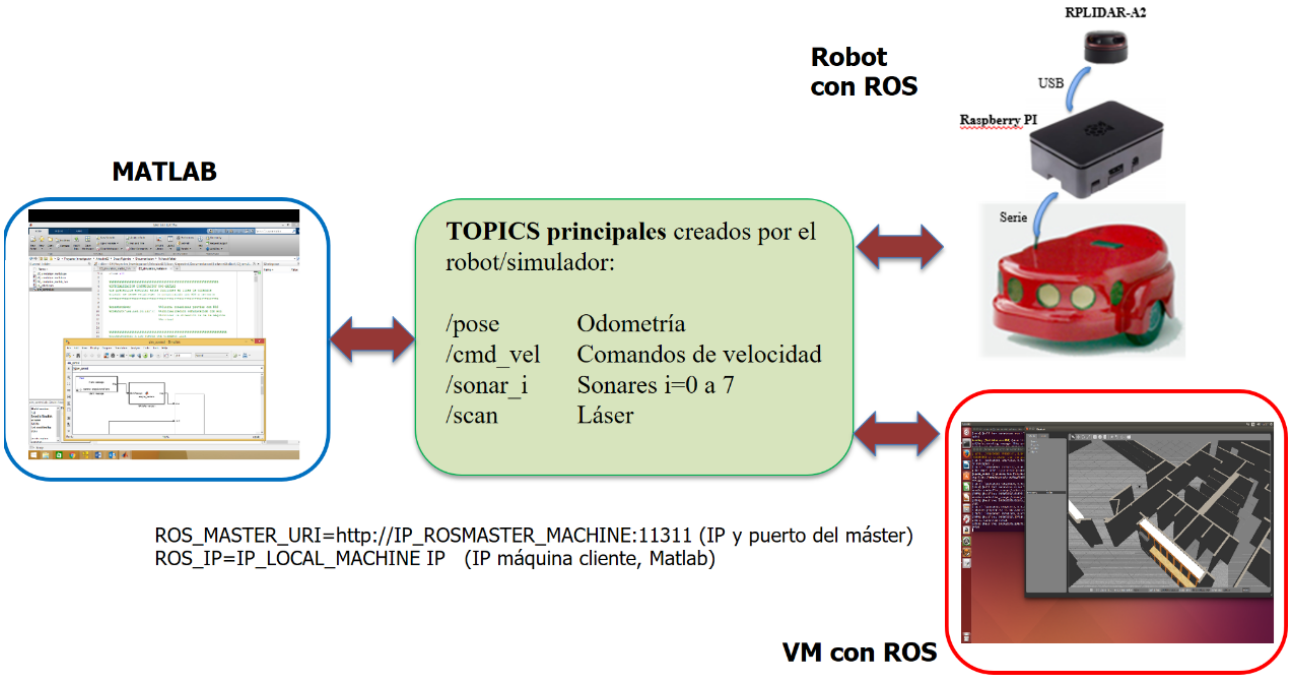

Figura 3.3: Configuración distribuida con robots reales.

En el arranque del robot, el sistema operativo ejecuta un script que configura y lanza de forma automática un Master de ROS, así como los drivers de ROS necesarios para controlar el robot y acceder a los datos de los sensores. Estos drivers son:

- p2os: explicado en el apartado 2.2.0.4, gestiona la comunicación con el microcontrolador para enviar comandos de velocidad y para leer la odometría y los sónares. Algunos de los topics más importantes creados por este driver son /pose, /cmd_vel, /cmd_motor_state y /sonar_i.
- rplidar: accede a los datos del láser dejándolos en el topic /scan. Para probar los scripts de teleoperación y lectura de sensores que se programaron en el apartado anterior únicamente hay que modificar el fichero ini_simulador.m por ini_amigobot.m, que se suscriba a los topics creados por el robot:

En este caso, los Subscribers se suscriben a los siguientes topics:

- Odometría: /pose
- Láser: /scan
- Sonar: /sonar_i

Además, en el caso del robot real es necesario publicar en un nuevo topic para habilitar los motores: */cmd_motr_state*, siendo el mensaje de tipo *std_msgs/Int32*. Para ello se debe escribir en el campo *Data* el valor 1. También hay que crear un Publisher para publicar los mensajes de velocidad como en el caso de la simulación, publicando en el topic */cmd_vel*.

Se comprobó que todos los ficheros de prueba creados para simulación funcionan correctamente en el robot real Amigobot así como en el Seekur, verificando la conectividad con las plataformas reales.

3.2 Mapeado del Entorno

En este apartado se desarrolla y explica el trabajo realizado para el mapeado del entorno. Se comparan tres algoritmos: uno de mapeado puro asumiendo posición actual conocida, el algoritmo de SLAM incluido en la Toolbox de Matlab y el nodo *SLAM_gmapping* de ROS.

3.2.1 Base teórica: SLAM

Una de las áreas de la robótica móvil en la que se está estudiando mucho en los últimos años está relacionada con la localización de los robots. Para el trabajo en ambientes exteriores existe una amplia gama de sensores GPS que permiten determinar de forma absoluta y cada vez con mayor precisión, la ubicación de un elemento. Sin embargo, en interiores, el problema de la localización absoluta con una cierta exactitud se convierte en algo crítico si se quiere dotar al robot de una cierta autonomía y precisión en su movimiento.

Para ello se pueden encontrar dos soluciones. La primera se basa en el estudio y desarrollo de algoritmos que tratan de localizar un elemento en un mapa a la vez que tratan de construir el propio mapa del entorno, dando lugar a los conocidos algoritmos SLAM (Simultaneous Location and Mapping). La segunda solución se basa en el refinamiento de la localización de un elemento en un mapa conocido. Por tanto, se puede definir SLAM como un concepto general para algoritmos que relacionan diferentes medidas de sensores para construir un mapa del entorno y crear posiciones estimadas.

El algoritmo SLAM de la *Robotics System Toolbox* usa medidas del láser e información de la odometría como entradas. La información de la odometría es una entrada opcional que da una posición estimada inicial para ayudar en la correlación de las medidas. Los algoritmos de *scan matching* relacionan las medidas con las añadidas previamente para estimar la posición relativa entre ellas y añadirlas al mapa de posición subyacente.

El gráfico de posición contiene nodos conectados por bordes que representan las posiciones relativas del robot. Los bordes especifican limitaciones en el nodo como una matriz de información. Para corregir las posiciones, el algoritmo las optimiza sobre el gráfico de posición completo cada vez que se detecta un lazo cerrado.

El algoritmo asume que los datos vienen de un robot, navegando en un entorno, que proporciona medidas del láser a lo largo de su camino. Por tanto, las medidas son primeramente comparadas con la medida más reciente para identificar posiciones relativas y son añadidas al gráfico de posición. Sin embargo, el algoritmo también busca lazos cerrados, los cuales identifican cuándo el robot está en un área previamente visitada.

3.2.2 Implementación con Matlab-ROS

Para estudiar este algoritmo se parte del ejemplo *OfflineSLAMExample.m*, el cual muestra cómo implementar un algoritmo de SLAM con una serie de sensores láser usando optimización de posición en el gráfico. El objetivo de este ejemplo es construir un mapa del entorno usando sensores y recuperar datos de la trayectoria del robot.

Para construir el mapa del entorno, el algoritmo de SLAM procesa las medidas de los sensores y construye un gráfico de posición que une esas medidas. El robot reconoce un lugar visto previamente conectando las medidas realizadas y puede establecer uno o más lazos cerrados a lo largo de su trayectoria. El algoritmo utiliza la información del lazo cerrado para actualizar el mapa y ajustar la trayectoria estimada del robot.

Primero carga una serie de datos consistentes en medidas del láser recogidos de un robot móvil en un entorno interior. Posteriormente se estudiará un algoritmo online en el que los datos no tengan que ser cargados previamente. Para guardar las medidas del láser durante el recorrido, usamos en ROS el comando *rosbag record /robot0/laser_1* y realizamos el recorrido por el mapa del pasillo, guardándose un fichero .bag con dichos datos. Este fichero puede ser leído desde Matlab. Las medidas importadas son del tipo *LaserScan*, sin embargo, para trabajar con el objeto *robotics.LidarSLAM* es necesario trabajar con medidas de tipo *LidarSCAN*, por lo que convertiremos todas las medidas al segundo tipo. Por otro lado, las medidas son tomadas a una frecuencia alta y no todas son necesarias para el SLAM, por tanto, se selecciona una de cada 40 medidas.

```
bag = rosbag('laser2.bag');
allLaserScans = readMessages(bag); %type LaserScan
allLaserScansT = allLaserScans';

for i=1:1:numel(allLaserScansT)
  scan(i) = lidarScan(allLaserScansT{1,i});
end

scan = scan(1:40:end);
```

Posteriormente hay que crear un objeto *robotics.LidarSLAM* e indicar la resolución del mapa y el rango máximo del sensor. Antes de esto, se realizó un estudio de la clase *robotics.LidarSLAM*. Esta se utiliza para realizar una localización y mapeado simultáneos usando como entrada las medidas de los sensores. El algoritmo de SLAM coge estas medidas y las conecta a un nodo en un gráfico de posición subyacente. Después relaciona las medidas usando *scan matching*. También busca lazos cerrados, donde las medidas superponen regiones mapeadas previamente, y optimiza las posiciones de los nodos en un gráfico de posición. Las propiedades y métodos principales son:

Propiedades:

- ***PoseGraph***: gráfico de posición subyacente que conecta las medidas. Al añadir las medidas a LidarSLAM se actualiza este gráfico. Cuando se encuentra un lazo cerrado, el gráfico de posición es optimizado usando OptimizationFcn.
- ***MapResolution***: resolución del mapa de ocupación, en celdas por metro.
- ***MaxLidarRange***: rango máximo del lidar, especificado en metros.
- ***OptimizationFcn***: función de optimización del gráfico de posición.
- ***LoopClosureThreshold***: (100 por defecto). Umbral para aceptar lazos cerrados. Un valor alto corresponde a una mejor unión.
- ***LoopClosureSearchRadius***: radio de búsqueda para detectar lazos cerrados. Incrementar este radio afecta al desempeño aumentando el radio de búsqueda.
- ***LoopClosureMaxAttempts***: número de intentos para encontrar lazos cerrados. Incrementar este valor afecta al desempeño aumentando el tiempo de búsqueda.
- ***LoopClosureAutoRollback***: permite la reducción automática de lazos cerrados, especificándose como true o false. El objeto SLAM comprueba el error residual recibido de la OptimizationFcn. Si detecta un cambio repentino en el error residual y su propiedad es true, rechaza el lazo cerrado.
- ***OptimizationInterval***: número de lazos cerrados aceptados para llevar a cabo la optimización.
- ***MovementThreshold***: mínimo cambio en la posición requerida para procesar las medidas, especificada como un vector [translation rotation]. El cambio de posición relativa para añadir una nueva medida viene dado como [x y theta]. Si la traslación en la posición xy o la rotación de theta excede estos umbrales, el objeto LidarSLAM acepta la medida y añade una posición al gráfico.

Métodos:

- ***addScan***: añade una medida al mapa.
- ***copy***: copia el objeto LidarSLAM.

- ***removeLoopClosures***: borra lazos cerrados del gráfico de posición.
- ***scansAndPoses***: extrae medidas y sus posiciones correspondientes.
- ***show***: muestra medidas y posiciones del robot.

slamObj = LidarSLAM(mapResolution,maxLidarRange) crea un objeto LidarSLAM, el cual se usa para añadir y comparar repetidamente medidas de los sensores y construir un gráfico de posición para la trayectoria del robot. Para establecer el algoritmo de SLAM, es necesario especificar el rango del láser, la resolución del mapa, el umbral de cierre de lazo y el radio de búsqueda. El rango del láser debe ser ligeramente menor que el rango máximo (8 metros), ya que las lecturas del láser son menos precisas cerca del rango máximo. La resolución del mapa debe ser 20 celdas por metro, lo que da una precisión de 5 cm. Los parámetros del lazo cerrado fueron establecidos empíricamente. Usar un umbral mayor de lazo cerrado ayuda a rechazar falsos positivos en el proceso de identificación de lazos cerrados. Sin embargo, hay que tener en cuenta que una combinación de puntuación alta puede ser también una mala combinación. Por ejemplo, medidas recogidas en un entorno que tiene características similares o repetidas son más propensas a producir falsos positivos. La propiedad *LoopClosureSearchRadius* indica el radio de búsqueda para la detección de lazos cerrados, especificado como un escalar. Cuando este radio es elevado, el tiempo de búsqueda es mayor.

Una vez creado el objeto se usa un bucle para añadir medidas, las cuales son comparadas con las anteriores por *scan matching*. Para mejorar el mapa, el objeto optimiza el gráfico de posición cada vez que detecta un cierre de lazo. Cada diez medidas se muestran las posiciones y medidas almacenadas para que pueda ser comprobado visualmente.

Después de añadir todas las medidas al objeto de SLAM, se construye un mapa *robotics.OccupancyGrid* llamando a la función *buildMap* pasando como parámetros las medidas y posiciones almacenadas, la resolución de mapa y el rango máximo del láser.

```
for i = 1:numel(scan)

  addScan(slamObj,scan(i));

  if rem(i,10) == 0
    show(slamObj)
  end

end

[scansSLAM,poses] = scansAndPoses(slamObj);
occGrid = build(scansSLAM,poses,resolution,maxRange);
figure
show(occGrid)
title('Occupancy Map of Hall')
```

Tras esta primera prueba offline, se realizó un nuevo script en el que las medidas son recogidas online, es decir, son obtenidas directamente del robot según este se desplaza por el entorno, lo cual da una visión más realista de lo que significa el método de SLAM. Para ello debemos conectarnos a ROS y suscribirnos a los nodos correspondientes al láser y a la odometría, recibiendo datos de estos en cada iteración.

```
while(1)

scanMsg = receive(laserSub);
odompose = odom.LatestMessage;
ranges = double(scanMsg.Ranges);
angles1 = double(scanMsg.readScanAngles);
angles = angles1;

% Crear objeto lidarScan
scans = lidarScan(scanMsg);
```

El resto del procedimiento a seguir es similar al explicado anteriormente. Se añaden medidas incrementalmente al objeto *slamAlg*. Los lazos son detectados automáticamente según se mueve el robot, optimizándose el gráfico de posición cada vez que se detecta uno. La salida *optimizationInfo* tiene un campo, *IsPerformed*, que indica cuando ocurre esta optimización. Si este campo está activo y el flag de *firstLoopClosure* también, se muestra por pantalla el primer lazo cerrado detectado, como se muestra en la figura 3.4.

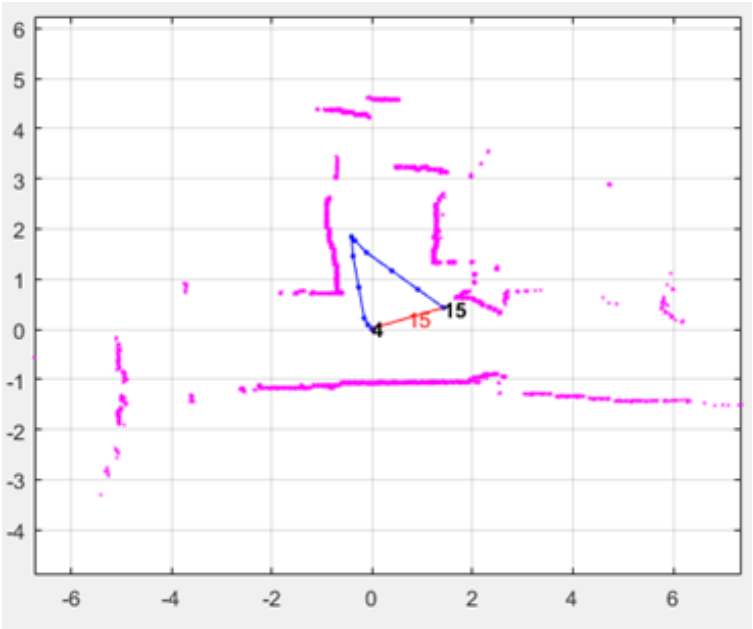

Figura 3.4: Algoritmo SLAM en simulación: Primer lazo cerrado.

Una vez han sido añadidas todas las medidas, se muestra el mapa final:

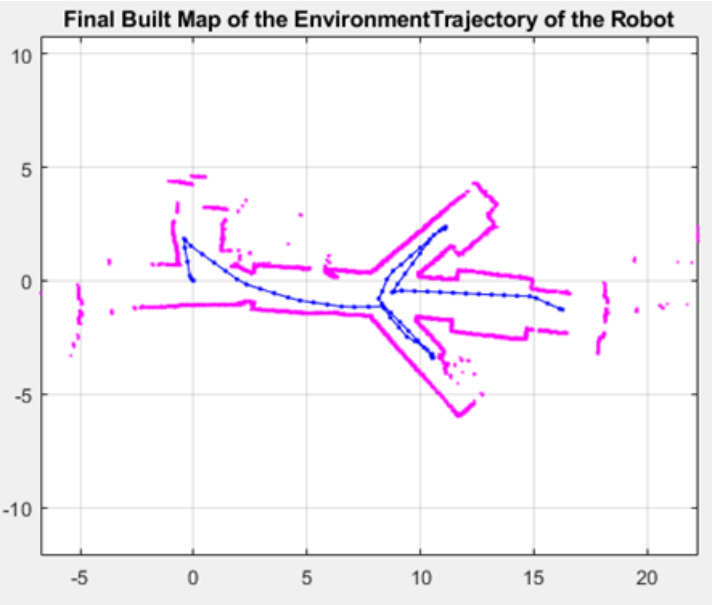

Figura 3.5: SLAM en simulación: Mapa final

Las medidas optimizadas y posiciones pueden ser usadas para generar un *robotics.OccupancyGrid*, el cual representa el entorno en una rejilla de ocupación. Finalmente se superpone la imagen con las medidas y el gráfico de posición al mapa original, observándose cómo el mapa obtenido es similar al anterior.

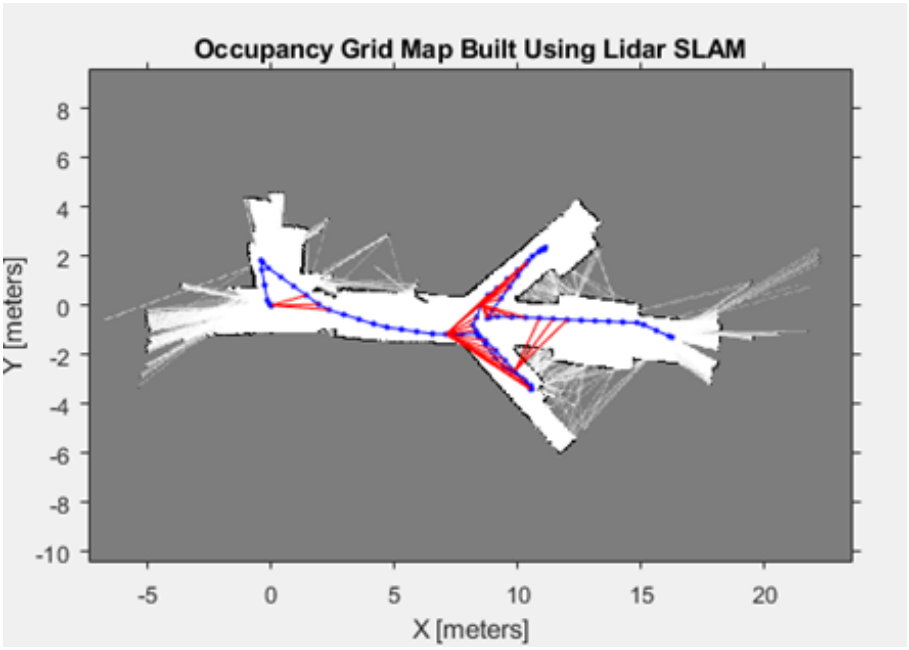

Figura 3.6: SLAM en simulación: Mapa final del tipo Occupancy Grid.

3.2.3 Resultados

En este apartado se analizarán los resultados obtenidos tanto en simulación como en las dos plataformas reales. Las pruebas se realizaron en el pasillo central de la segunda planta del lado oeste del edificio politécnico de la Universidad de Alcalá. Para las pruebas en simulación, se usó un mapa de dicho pasillo obtenido mediante el algoritmo *gmapping*, explicado en la sección 3.2.4.2.

Para la prueba offline en simulación (*Offline_SLAM.m*) los datos fueron recogidos de la trayectoria del robot a lo largo del mapa recién mencionado. Estos se leen desde Matlab con la instrucción *rosbag(_laser2.bag)*. Se observa que el mapa obtenido coincide perfectamente con el real, incluyendo detalles como las puertas abiertas. En la figura 3.7 se muestra el objeto Slam, actualizado cada 10 medidas, y el mapa de ocupación final con el recorrido seguido por el robot, así como el cierre de lazos.

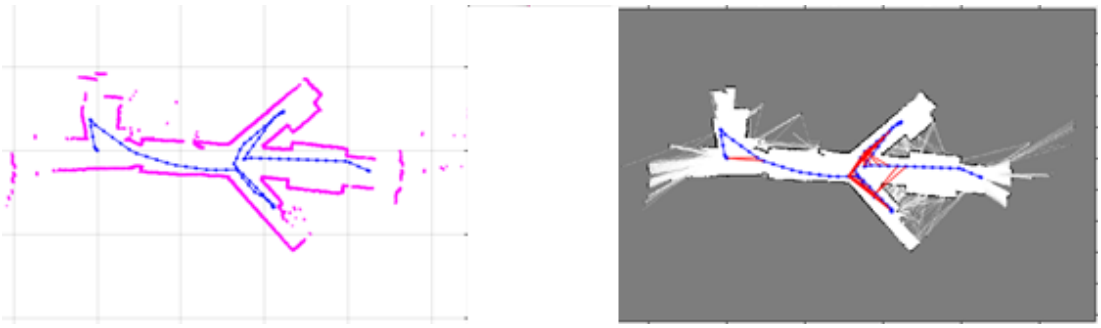

Figura 3.7: SLAM: Resultados en simulación.

Tras esta primera prueba, se pasó a probar el script online (*Online_SLAM.m*). El algoritmo va recogiendo medidas según teleoperamos el robot en el simulador. Para ello debemos suscribirnos a los topics */robot0/laser_1* y */robot0/local_odom*. El mapa obtenido es similar al mostrado en la figura 3.7, sin observarse ninguna diferencia aparente.

Tras comprobar que los dos scripts funcionan correctamente con simulación, probamos con el robot real Amigobot. Realizamos un recorrido por el pasillo y guardamos los datos del láser en *laser_robot2.bag*. En la figura 3.8 se muestra el objeto de SLAM; es decir, las medidas y posiciones actualizadas cada 10 medidas, así como el mapa de ocupación final con la trayectoria seguida y el cierre de lazos.

Comparando el mapa con el obtenido mediante simulación se observa una ligera diferencia, siendo este menos preciso que el anterior en la parte inicial del mapa. Sin embargo, según avanza el robot, se observa que el mapa se adecua muy bien al mapa real. Esto puede deberse a una mala toma de datos al guardar el .bag. Sin embargo, tras varias pruebas se observó el mismo comportamiento.

Al igual que en simulación, se realizó una segunda prueba, obteniendo los datos del láser mientras el robot recorría el entorno. El único cambio en el código es el nombre de los nodos a los que hay que subscribirse, siendo en este caso /scan y /pose para el láser y la odometría respectivamente. De nuevo, el resultado obtenido es similar al obtenido a partir de los datos guardados en el .bag.

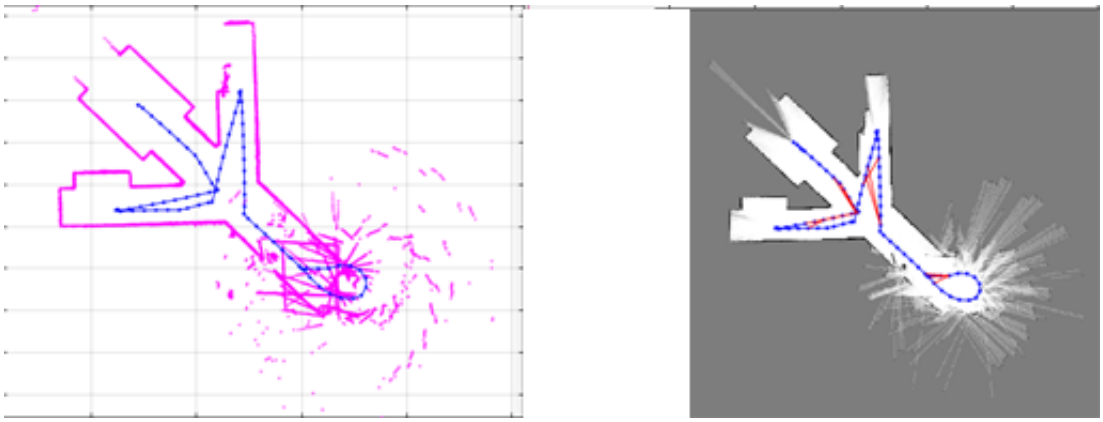

Figura 3.8: Resultados en Plataforma Amigobot

Se realizó una última prueba con el robot Seekur. En este caso el recorrido fue llevado a cabo en la segunda planta del lado oeste de la Escuela Politécnica, debido al mayor tamaño del robot. La única diferencia en programación fue la creación de la función *truncar_lidar*. Debido a la posición del láser en el Seekur, se veía la parte de atrás del propio robot, por ello se creó dicha función. En ella se eliminan partes de las medidas obtenidas por el láser, de forma que sólo nos quedamos con la información que nos interesa.

```
function lidar_truncado=truncar_lidar(lidar)

alfa=90*pi/180;
x=2*alfa/lidar.AngleInrement;
n=length(lidar.Ranges);
ini=(n/2)-(x/2);
fin=(n/2)+(x/2);

%Mensaje con el lidar truncado
lidar_truncado = rosmessage('sensor_msgs/LaserScan');

lidar_truncado.Header=lidar.Header;
lidar_truncado.AngleMin=-alfa;
lidar_truncado.AngleMax=alfa;
lidar_truncado.AngleIncrement=lidar.AngleIncrement;
lidar_truncado.TimeIncrement=lidar.TimeIncrement;
lidar_truncado.ScanTime=lidar.ScanTime;
lidar_truncado.RangeMin=lidar.RangeMin;
lidar_truncado.RangeMax=lidar.RangeMax;
lidar_truncado.Ranges=lidar.Ranges((ini:fin));
lidar_truncado.Intensities=lidar.Intensities((ini:fin));

end
```

En las figuras 3.9 y 3.10 y se muestran los mapas obtenidos.

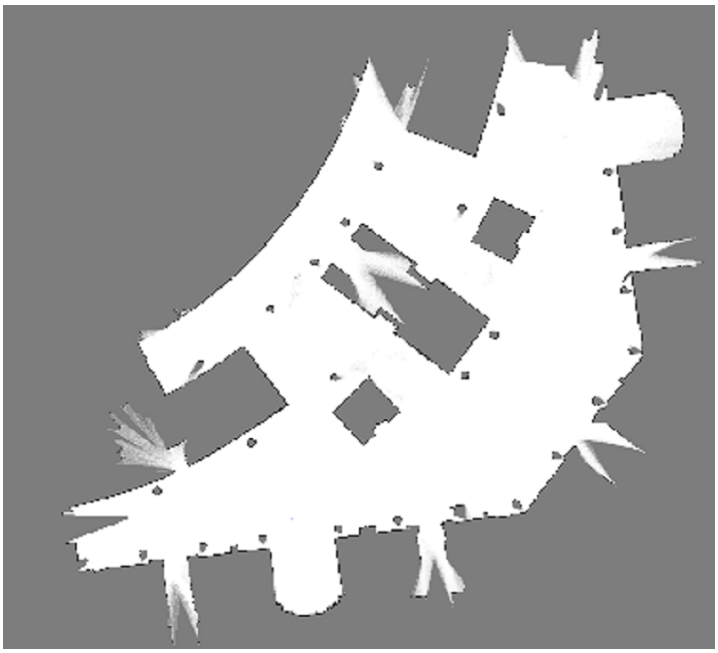

Figura 3.9: Resultados en Plataforma Seekur

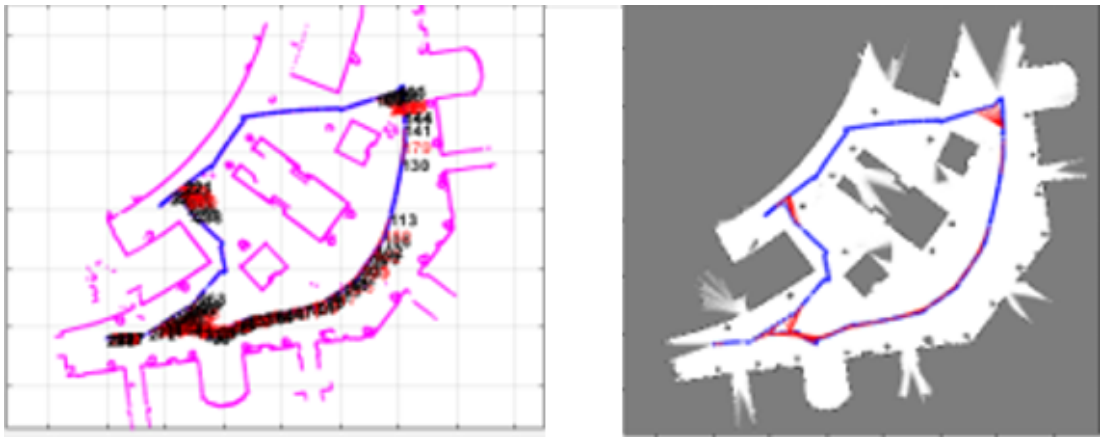

Figura 3.10: Resultados en Plataforma Seekur

De nuevo, se observa que el mapa obtenido es muy preciso. En conclusión, podemos decir que se trata de un algoritmo de SLAM muy robusto, ya que es muy exacto y presenta pocos fallos. Una alternativa muy recomendable frente al nodo *slam_gmapping* de ROS, el cual presenta múltiples fallos y poca precisión.

3.2.4 Comparación con nodo slam_gmapping y mapeado con posiciones conocidas.

Además del algoritmo de SLAM explicado se prueban y comparan otras dos técnicas para obtener un mapa de ocupación del entorno utilizando el láser y la odometría.

3.2.4.1 Mapeado con posiciones conocidas.

En primer lugar se prueba una técnica de mapeado puro, en el que se asume que la posición actual es conocida, lo cuál no es cierto en un caso real. Esta consiste en convertir las medidas de distancia obtenidas del láser en coordenadas cartesianas relativas al robot e insertar los obstáculos detectados en una rejilla de ocupación respecto a la posición actual del robot.

Esta técnica se programó en el script *MappingWithKnownPoses.m*, testeada en el simulador STDR. La teleoperación del robot por el entorno para ir obteniendo el mapa se realizó mediante el nodo de ROS *teleop_twist_keyboard.py* . Con el fin de que la simulación fuera lo más realista posible, se configuró un error en la odometría del robot dentro del nodo *aux_files*. Para ello creamos un nodo que a su vez crea el topic */local_odom*, el cual localiza al robot en (0,0) como posición inicial y crea un error acumulativo por defecto, el cual es bastante realista.

Primero, se editó *amigobot.launch*[2] (fichero que permite lanzar el simulador con el robot amigobot), usando la sentencia *<remap from=_tf to=tf_sim/>* en todas las líneas de nodos. Además este archivo contiene dos *include*, de esa manera incluimos esta misma sentencia también en estos ficheros.

/tf[3] es un topic especial de ROS que está siempre presente y sirve para relacionar los distintos marcos de coordenadas que existen en el sistema. Es gestionado por un paquete llamado TF que incluye múltiples utilidades para trabajar con los marcos de coordenadas y gestionar sus transformaciones. La relación entre los sistemas de coordenadas, incluso cuando es constante a lo largo del tiempo, ha de ser publicada de forma continua en el topic */tf* para que el resto de nodos pueda utilizar esta información. Usualmente, cada topic tiene información referida a un sistema de coordenadas. Este sistema de coordenadas se encuentra indicado explícitamente en la cabecera del tipo de datos (campo *frame_id*).

Lo que hacemos con la sentencia *<remap from=_tf to=tf_sim/>* es desechar la información del nodo *tf*, la cual nos localiza al robot inicialmente en (4,8), dado que da la posición relativa a la esquina inferior izquierda del mapa del simulador STDR. En el archivo *aux_files* (en el que creamos anteriormente el nodo *local_odom*), se crea otro nodo denominado *tf_repub* el cual coge la información que sí nos interesa del nodo *tf*; esta es, la relación entre los marcos de coordenadas del láser y los sonar respecto al robot. Podemos comprobar, usando la herramienta *rqt*, el árbol de transformadas de *tf_sim* (3.11) y *tf* (3.12).

[2]Detallado en la sección A.5 dentro del Manual de Usuario.
[3]Una explicación más detallada de este topic se puede encontrar en el Manual de Usuario, en la sección A.3.3.

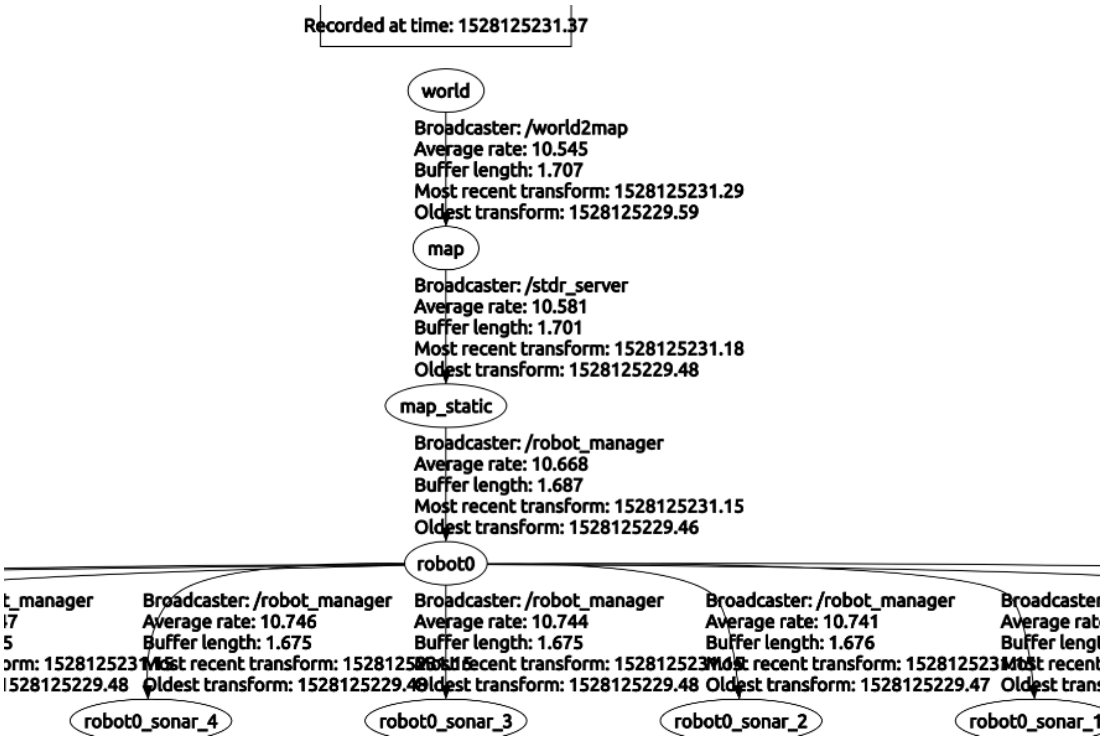

Figura 3.11: Árbol de transformadas tf_sim

En la figura 3.11 se puede comprobar que existe un marco base de coordenadas llamado *world* relacionado con el marco *map*. Esta relación la realiza el nodo *world2map* el cual publica una transformada estática entre ambos. A su vez el marco *map* está relacionado por una transformada estática enviada por el simulador con el marco de coordenadas *map_static*. Los marcos de coordenadas del mapa y el robot están relacionados por la odometría del mismo, relación que cambia conforme se mueve el robot y que está publicada por el simulador. También existen una serie de transformadas entre el robot y sus diferentes sensores, que son publicadas por el simulador.

En la figura 3.12, los marcos de coordenadas *world*, *map* y *map_static* desaparecen, teniendo en este caso el campo de coordenadas *odom* el cual ofrece las posiciones del robot relativas a su posición inicial, igual que ocurre con robots reales. Además de introducir un error acumulativo, como se indicó anteriormente.

Dentro del script de Matlab se crea un mapa de ocupación con el objeto *robotics.OccupancyGrid* de alta resolución para capturar las medidas del láser. Después, dentro del bucle de control se llevan a cabo las siguientes tareas:

- Se reciben los datos del láser, tras suscribirnos a este, y se utiliza la función *getTransform* para realizar la transformación del sistema de coordenadas de */odom* al de */robot0*.
- Se obtiene la posición del robot y la orientación en forma de cuaternio, que se convierte a ángulos de Euler para obtener la posición en forma de vector [x y yaw].

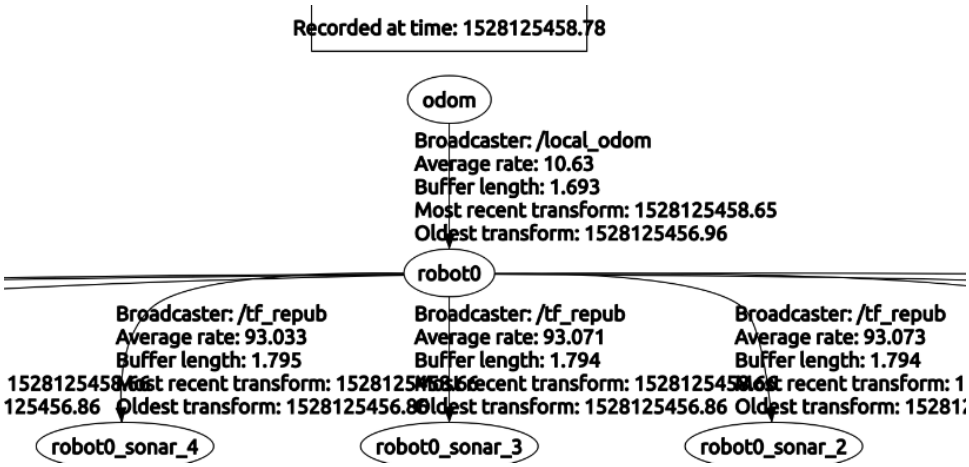

Figura 3.12: Árbol de transformadas tf

- Pre–procesamiento de los datos del láser. El simulador devuelve rangos NaN para aquellos haces del láser que no chocan con ningún obstáculo. En este caso, se reemplazan por el valor máximo (8m).
- Se insertan las observaciones del láser en el mapa con *insertRay* y se visualiza el mapa cada 50 actualizaciones.

El mapa obtenido es el mostrado en la figura 3.13.

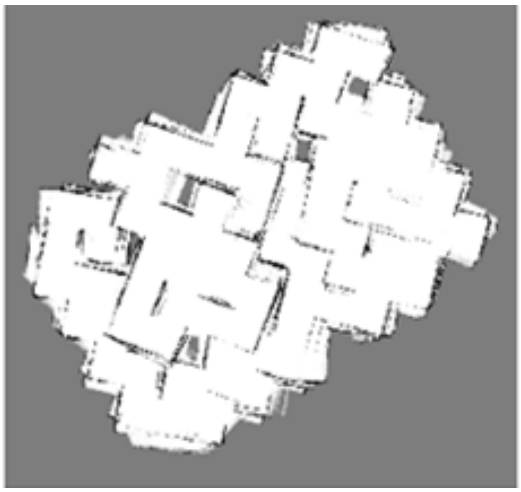

Figura 3.13: Resultado del mapeado asumiendo posiciones conocidas.

Se realiza así mismo una prueba configurando un error nulo para la odometría, y en el que las posiciones se obtienen respecto al origen de coordenadas del simulador. Por tanto, en *getTransform*, la transformada se realizará entre el sistema de coordenadas del mapa (*/map*) y del robot (*/robot0*). Se observa una notable diferencia en la calidad del mapa, como puede observarse en la figura 3.14.

Sin embargo, el primero es mucho más realista, ya que se obtienen posiciones relativas a la posición inicial del robot (/odom), no respecto al origen de coordenadas del simulador (/map), además de ir introduciendo un error acumulativo. Este es el principal problema del SLAM.

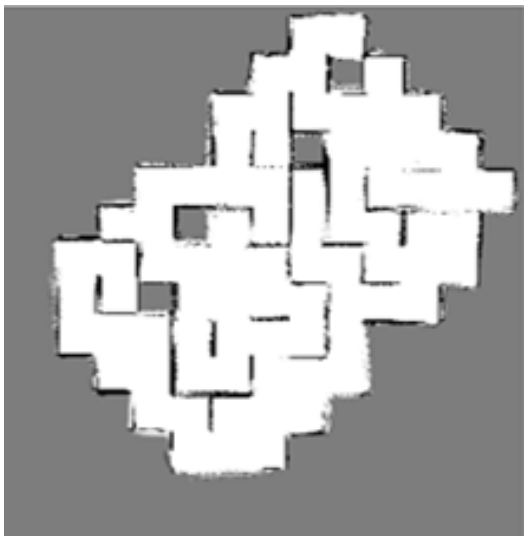

Figura 3.14: Resultado mapeado puro con error nulo en simulación.

Los pequeños errores se van acumulando produciendo graves errores globales, sobre todo en los cierres de lazo. Esto es lo que ocurre en robots reales, la odometría toma como punto de origen la posición inicial de robot y siempre hay un error, ya que no existen sensores perfectos. Existe un error tanto de mapeado como de localización, el cual se va acumulando, creando un falseo en el mapa.

Como se puede apreciar, los resultados son mucho peores que los obtenidos mediante mapeado y localización simultáneos.

3.2.4.2 Nodo slam_gmapping

La siguiente técnica a comparar sí que se trata de un algoritmo de SLAM, pero esta vez realizado a través del nodo *slam_gmapping* de ROS, [53], configurándolo y lanzándolo a través de un fichero .launch.

El paquete *gmapping* realiza las siguientes suscripciones y publicaciones:

Suscripciones:

- `tf`: transformadas necesarias para relacionar los frames del láser, el robot y la odometría.
- `scan`: sensor láser utilizado

Publicaciones:

- `map`: mapa creado.
- `map_metadata`: datos del mapa.
- `entropia`: valor de la entropía de la distribución. A mayores valores más incierto es el mapa.

Primero se crea el fichero *gmapping_Simulator.launch*[4] con todos los parámetros necesarios adaptados al robot (frames y distancias de los sensores) para que pueda ejecutarse en el simulador. Los únicos parámetros que hay que modificar respecto a su valor por defecto son *base_frame* por */robot0*, *maxUrange* y *maxRange*. Además hay que remapear el topic */scan* para que se adapte al del simulador, llamado */robot0/laser_1*, así como el topic */map* en el que el nodo slam_gmapping publica el mapa generado a otro nombre distinto, ya que el topic */map* ya existe, creado por el simulador y con información completa del mapa. En este caso lo remapeamos a otro topic llamado */gmapping_map*. Recorrimos el entorno haciendo uso del controlador por teclado y generamos un mapa lanzando el archivo creado y haciendo uso del visualizador *RVIZ* explicado en la sección 2.2.0.3.

Creamos otro fichero .launch llamado *gmapping_RealRobot.launch* para realizar pruebas utilizando el robot real, lo único que hay que modificar respecto al fichero anterior son los remapeados y el *base_frame*: el topic */scan* en este caso tiene el mismo nombre que el topic en el que se publican los datos del láser; además, no es necesario remapear el topic */map*, puesto que no estamos usando el simulador; el *base_frame* toma ahora el valor de *base_link*, ya que en el árbol de transformadas se observa que el marco de referencia del robot toma ahora este nombre. Recorrimos el pasillo y guardamos el mapa generado, mostrado en la figura 3.15.

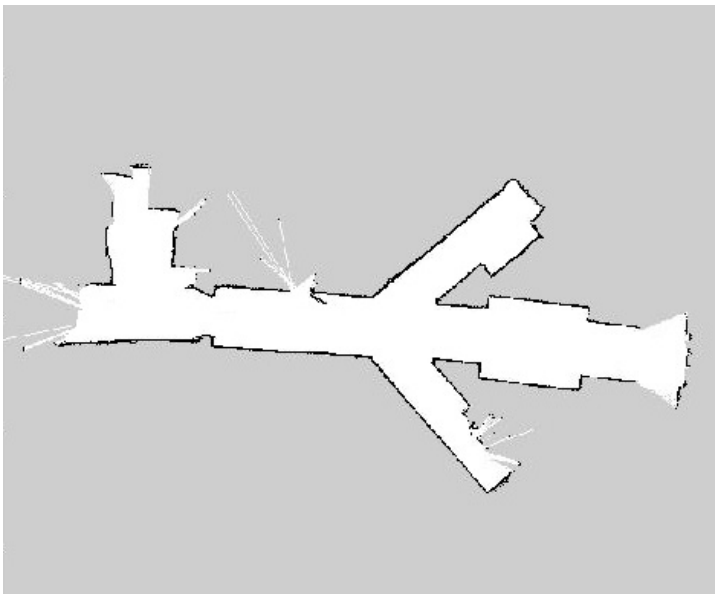

Figura 3.15: Resultado gmapping.

Tanto en simulación como con el robot real Amigobot, los resultados finales son bastante buenos. Sin embargo, se observa que el algoritmo de SLAM programado en Matlab responde siempre bien, mientras que el gmapping falla en un gran número de ocasiones. Con el robot Seekur, no se consigue obtener un mapa tan preciso como el obtenido con el algoritmo de Matlab. Este presenta muchos fallos y zonas discontinuas como se observa en la figura 3.16. Cuando tratamos de pasar por estas zonas para recoger mejores medidas y reconstruya el mapa, este pierde totalmente su consistencia.

[4]La ruta a los directorios en los que se encuentran todos los archivos de los que se habla en este capítulo se encuentran dentro del anexo Planos, en la sección C.2.

El problema con este algoritmo es que hay que tratar de realizar el mapa correctamente en un solo recorrido ya que, al pasar repetidas veces por la misma zona, deja de localizarse bien y se pierde el mapa. Esto no ocurría con el código de Matlab, en el que podías pasar repetidas veces por la misma zona para reforzar las medidas tomadas en estas y construir un mapa más consistente.

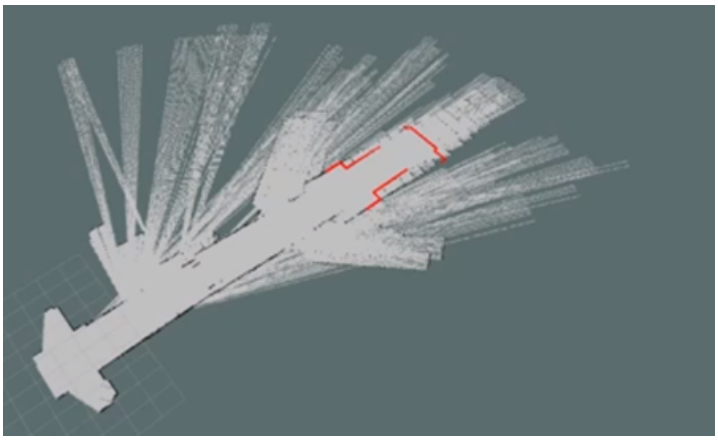

Figura 3.16: Resultado gmapping en plataforma Seekur

Por tanto, podemos concluir que el algoritmo de SLAM de la toolbox de Matlab es mucho más eficaz que el realizado con el nodo *gmapping* de ROS. También se verifican las ventajas del mapeado y localización simultáneos, frente a un mapeado puro, donde los resultados son mucho más pobres.

3.3 Localización

Pasando a la segunda solución de las mencionadas para resolver el problema de la localización en interiores, se propone una implementación del *Método de Montecarlo Adaptativo* (AMCL) para la localización robusta en un mapa previamente obtenido.

3.3.1 Base teórica: AMCL

El problema de la localización en robótica móvil se puede dividir en dos grandes grupos: la localización global, para estimar la posición absoluta global del robot, y la localización local, para hacer un seguimiento de su posición de forma local.

La localización global se define como la ubicación del robot dentro de un espacio conocido, siendo esta la única información disponible para el robot. Si se considera que el robot se encuentra localizado dentro de dicho mapa, se pasa a realizar de forma local un seguimiento de su posición. Esta subdivisión permite utilizar primeramente el posicionamiento global para que, haciendo uso del mapa, sea posible planificar una trayectoria para el robot, mientras que el seguimiento local le permite realizar una navegación eficiente. De esta manera se tienen en cuenta imprevistos no presentes en el mapa inicial, como obstáculos móviles. Los algoritmos de planificación global y local serán tratados más en detalle en los apartados 3.4 y 3.5.

El método de localización estudiado en este apartado se denomina Localización de Monte Carlo (MCL) [31]. El objetivo es la estimación de la posición en un instante de tiempo con el conocimiento de todas las medidas obtenidas por el robot hasta el instante t. Se trabaja con un vector de estado que contiene la posición y orientación del robot. Se define así la probabilidad de que una posición s_t sea la actual del robot en el instante t. El cálculo de la probabilidad para todas las posiciones posibles proporciona la función de densidad de la probabilidad de la posición del robot en un momento dado. Para calcular la función de densidad de la probabilidad se parte de un estado inicial y se realiza un cálculo en dos fases de forma iterativa: fase de predicción y fase de estimación.

Cada partícula tiene la forma p=(s,w), donde s es la posición (x,y,θ) y w el peso de la partícula, la probabilidad de ser la posición real del robot. Se realiza una primera etapa de predicción, donde se genera un nuevo conjunto de partículas a partir del anterior. Cada nueva partícula se genera en dos pasos: se selecciona aleatoriamente una partícula anterior, con probabilidad determinada por el peso de dichas partículas y se propaga dicha partícula a su nueva posición utilizando el modelo de movimiento p(s'|s,a), al cual se le aplica un pequeño ruido gaussiano que tiene como objetivo modelar el error de un desplazamiento calculado exclusivamente a través de la odometría del robot. La segunda etapa es la de estimación, donde se obtiene una medida con los sensores, recalculando así los pesos de cada una de las partículas $w_{ti} = \eta p(o_t|s_{ti})$. Este cálculo en dos fases se repetirá indefinidamente. El ruido gaussiano en la fase de predicción hará que las partículas clonadas se dispersen fomentando que las partículas se agrupen en torno a áreas de interés, en lugar de estar repartidas por todo el mapa. Finalmente las partículas convergerán a una zona concreta del mapa determinando la posición del robot real.

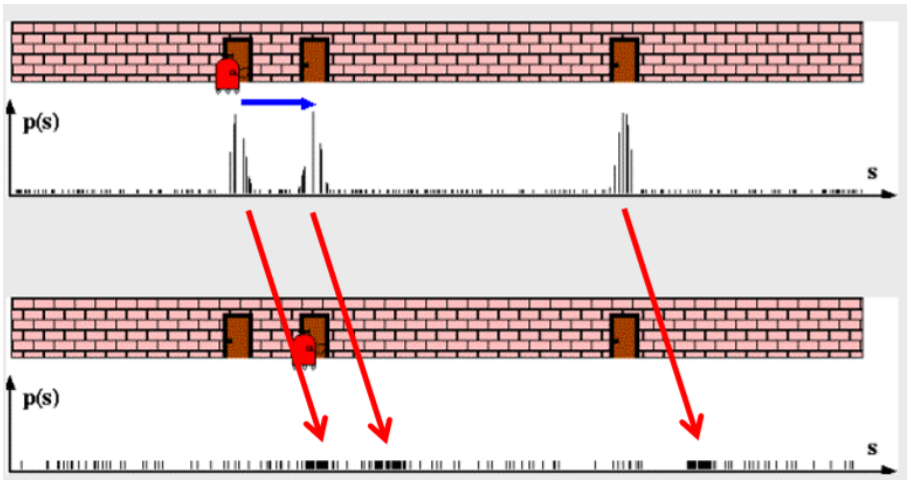

Figura 3.17: Ejemplo de fase de predicción.

Los métodos probabilísticos en general, son muy robustos en navegación, ya que permiten una representación constante y en tiempo real de la incertidumbre de la posición global. Estos permiten una localización más precisa que con modelos basados en subdivisión del mapa en celdas fijas, permitiendo también una fácil implementación. Sin embargo, este método puede dar lugar a falsas localizaciones, especialmente en mapas simétricos, donde es imposible diferenciar la posición del robot únicamente con información geométrica. Además, este método requiere una mayor potencia computacional para realizar los múltiples cálculos. Pero esto no es un problema

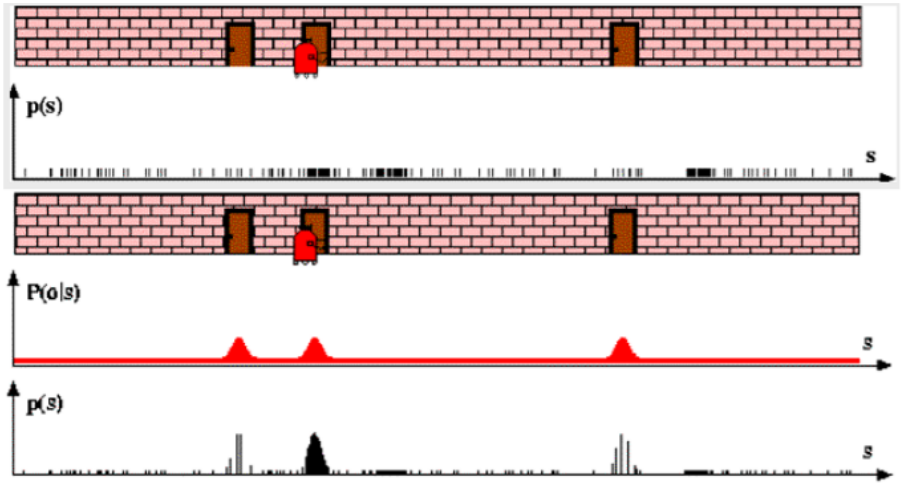

Figura 3.18: Ejemplo de fase de estimación.

importante, ya que cada vez existen más sistemas empotrados de tamaño reducido con una alta capacidad de cómputo. Así mismo, se puede conectar de manera inalámbrica a un PC remoto al que delegar dichos cálculos.

3.3.2 Implementación con Matlab-ROS

La Robotics System Toolbox incluye la clase *robotics.MonteCarloLocalization*, que permite localizar un robot utilizando datos del láser y la odometría sobre un mapa conocido a priori. En este caso el mapa ya ha sido obtenido en ROS mediante el nodo *slam_gmapping*, y guardado utilizando el nodo *map_saver*. El mapa mostrado en la figura 3.19 es importado en Matlab en formato de rejilla de ocupación (*robotics.OccuncyGrid*).

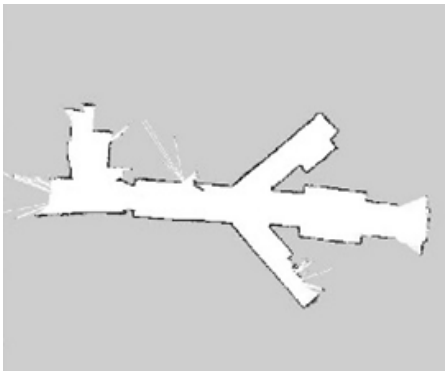

Figura 3.19: Mapa obtenido mediante nodo slam_gmapping.

Robotics.MonteCarloLocalization crea un objeto de localización Monte Carlo (MCL). El algoritmo de MCL es usado para estimar la posición y orientación de un robot en su entorno usando un mapa del entorno conocido, los datos del láser y de la odometría. Utiliza un filtro de partículas para estimar la posición del robot. Las partículas representan la distribución de estados posibles del robot, donde cada partícula representa un estado posible del robot. Estas convergen alrededor de una única localización según se mueve el robot por el entorno y localiza

diferentes partes del entorno usando sensores. Un sensor de odometría mide el movimiento del robot.

Antes de empezar la programación de la aplicación, se hizo un estudio de las propiedades de esta clase, las cuales son nombradas a continuación:

- *InitialPose*: [0 0 0] por defecto, posición inicial.
- *InitialCovariance*: diag([1 1 1]) por defecto, valor de la covarianza inicial.
- *GlobalLocalization*: flag para empezar localización global. Si esta desactivado, se trata de localización local.
- *ParticleLimits*: [500 500] por defecto, mínimo y máximo número de partículas.
- *SensorModel*: campo de probabilidad del modelo de sensor.
- *MotionModel*: campo de probabilidad del modelo de movimiento.
- *UpdateThresholds*: [0.2 0.2 0.2] por defecto. Mínimo cambio requerido en alguna de las coordenadas [x y theta] para actualizar las partículas.
- *ResamplingInterval*: número de actualizaciones del filtro de partículas entre dos muestras.

Toma como entradas los datos de posición y del láser. Los datos del láser están referenciados a su propio marco de coordenadas, y el algoritmo transforma los datos de acuerdo a la propiedad *SensorModel.SensorPose*, especificada por el usuario. Si el cambio de posición es mayor que cualquiera de los umbrales de actualización, *UpdateThresholds*, especificados, las partículas son actualizadas y el algoritmo estima un nuevo estado del filtro de partículas. Las salidas del objeto son la posición estimada y su covarianza, así como el valor de *isUpdated*, que será *true* cuando el cambio de posición sea mayor que alguno de los umbrales. La posición se da en el marco de coordenadas del mapa especificado en la propiedad *SensorModel.Map*.

Primero, las partículas son propagadas basándose en el cambio de posición y el modelo de movimiento especificado en *MotionModel*. A estas partículas se les asigna pesos según la probabilidad de recibir la lectura del láser para cada partícula, basándose en el modelo del sensor especificado en *SensorModel*.

Para crear el modelo de movimiento se utiliza el objeto *OdometryMotionModel*, el cual modela un robot diferencial y utiliza los datos de odometría para estimar su movimiento. Este modelo asume que el robot realiza movimientos de rotación y traslación puros para desplazarse. La propiedad *Noise* es el ruido gaussiano o varianza del movimiento del robot. Por defecto es [0.2 0.2 0.2 0.2], siendo respectivamente cada uno de los elementos del vector:

- Error rotacional debido a movimiento rotacional.
- Error rotacional debido a movimiento traslacional.
- Error traslacional debido a movimiento traslacional.
- Error traslacional debido a movimiento rotacional.

Una vez modelado el movimiento, se hace uso del método de campo de probabilidad *LikelihoodFieldSensorModel* para hallar la probabilidad de percibir una serie de medidas comparando los puntos finales del rango de medida con el mapa de ocupación.

Para conseguir mejores resultados es necesario modelar el sensor lo más parecido posible al sensor real. La propiedad *SensorLimits* indica los rangos máximo y mínimo del sensor ([0.45 8]) y la propiedad *Map* representa el mapa de ocupación usado para representar el entorno del robot como una red de valores binarios, indicando 1 para los obstáculos y 0 para las zonas libres.

Por otro lado, la propiedad *SensorPose* hace referencia a la posición del sensor relativa al sistema de coordenadas del robot. Esto es necesario para transformar las lecturas del láser del sistema de coordenadas del sensor (*/robot0_laser1*) al sistema de la base del Amigobot (*/robot0*). Esta es la función de la siguiente porción de código:

```
tftree = rostf;
waitForTransform(tftree,'/robot0','/robot0_laser_1');
sensorTransform = getTransform(tftree,'/robot0', '/robot0_laser_1');

% Obtener los ángulos de Euler
laserQuat = [sensorTransform.Transform.Rotation.W sensorTransform.
   Transform.Rotation.X ...
sensorTransform.Transform.Rotation.Y sensorTransform.Transform.
   Rotation.Z];
laserRotation = quat2eul(laserQuat, 'ZYX');

% Setup de |SensorPose|
rangeFinderModel.SensorPose = ...
[sensorTransform.Transform.Translation.X sensorTransform.Transform.
   Translation.Y laserRotation(1)];
```

Una vez hecho esto, ya podemos crear e inicializar el objeto AMCL con todos sus métodos y propiedades explicadas anteriormente.

Para el desarrollo del algoritmo se creó un bucle en el que se van recibiendo los datos del láser y la odometría del robot, se conviertan los datos del láser al tipo *lidarScan* para pasárselo como parámetro al objeto AMCL. Este actualiza la posición estimada del robot y su covarianza en cada iteración utilizando los datos del sensor y la posición obtenida de la odometría. La posición ha de pasarse como un vector de 3 elementos: posición en x, y y el valor de yaw u orientación en el eje z, para lo cual es necesario convertir el cuaternio obtenido de la odometría a ángulos de euler. El objeto AMCl nos devuelve el valor de la variable *isUpdated*, la posición estimada y la covarianza.

En cada iteración se muestra una imagen con la posición estimada, las partículas y las lecturas del láser.

En el siguiente apartado se muestran los resultados obtenidos del algoritmo explicado.

3.3.3 Resultados

Se realizó un primer script para probar el algoritmo en el simulador STDR ejecutándose en ROS (*MonteCarloLocalization_Simulation.m*). La teleoperación del robot por el entorno se realizó mediante el nodo de ROS *teleop_twist_keyboard.py*.

Para ello se hizo uso del mapa obtenido en la sección de mapeado para que el robot se localizase en él. Haciendo una primera prueba de localización local, se configuró la posición inicial de manera que fuera conocida pero con cierto error, (3,7,0) en vez de la real (4,8,0) y se comprobó que el robot se localizaba perfectamente.

Cuando la posición inicial es desconocida, AMCL trata de localizar el robot sin conocer dicha posición, lo que recibe el nombre de localización global. El algoritmo inicial asume que el robot tiene la misma probabilidad, uniforme, de estar en cualquier sitio del mapa y genera partículas uniformemente distribuidas en el espacio. La localización global requiere significativamente más partículas que la local, por lo que especificamos un valor de [5000 50000]. Además, es necesario indicarlo en la propiedad *GlobalLocalization* como *true*.

En ambas pruebas se observó que el robot se localizaba perfectamente, aunque es más lento en el caso de la localización global.

Tras realizar estas dos pruebas, se adaptó el script para poder utilizarlo con el robot real (*MonteCarloLocalization_RealRobot.m*). La única diferencia es el nombre de los marcos de coordenadas */base_link* y */laser_frame* así como el nombre de los topics de odometría (*/pose*) y del láser (*/scan*). Además es necesario habilitar los motores y cambiar el mapa, por el del pasillo que se obtuvo en el apartado de mapeado, en el que la resolución es de 20 celdas por metro cuadrado, mientras que el del STDR es 33.

Utilizando localización global se comprobó que el robot se localizaba perfectamente en el mapa, como puede verse en las figuras 3.20 y 3.21 donde se observa el robot en verde, las partículas en azul y las lecturas del láser en rojo.

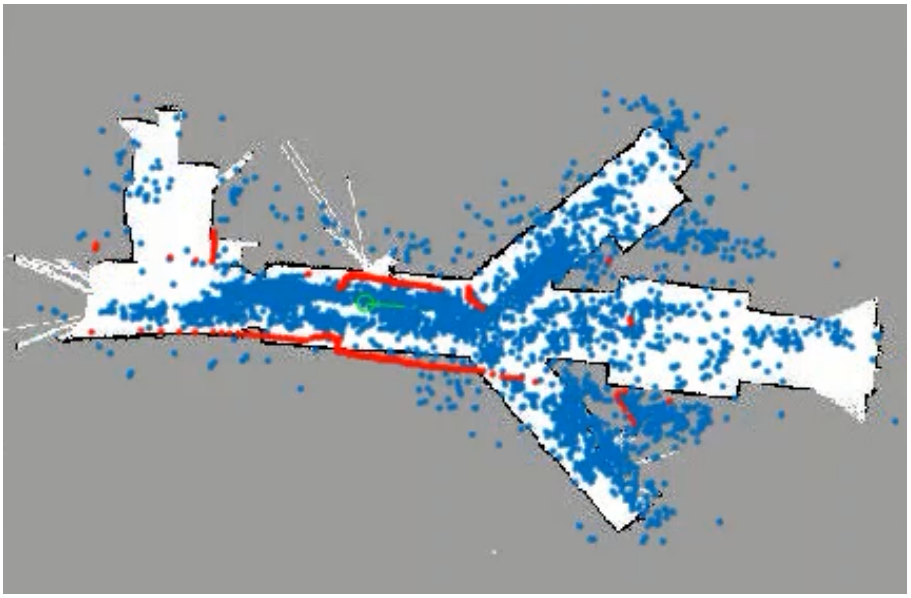

Figura 3.20: Localización global con Robot real: Inicio.

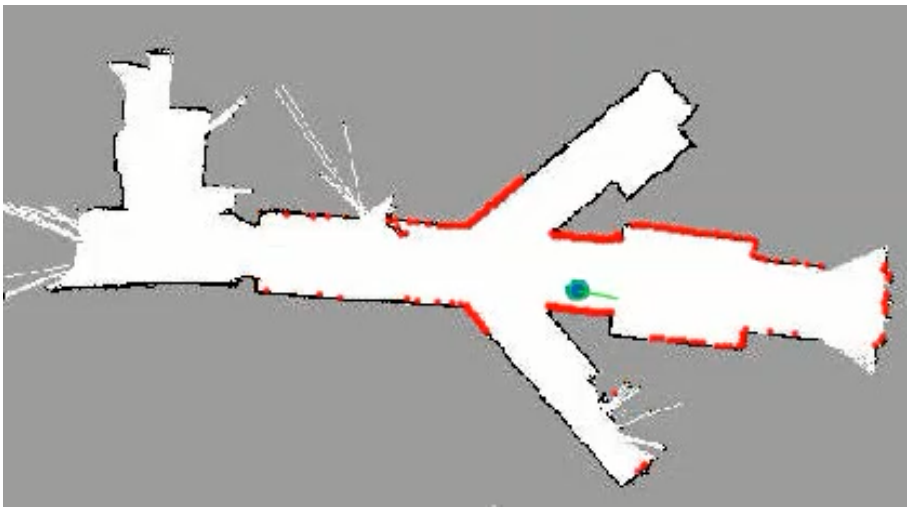

Figura 3.21: Localización global con Robot real: robot localizado.

En general, del estudio de este algoritmo se pueden derivar las siguientes ventajas e inconvenientes:

Ventajas:

- Resuelve los problemas de localización local y global.
- Admite cualquier tipo de modelo del sensor, dinámica de movimiento y distribución de ruido.
- Centra los recursos computacionales en las áreas del entorno en que hay más probabilidad de que se encuentre el robot.
- Precisión: no discretiza el estado.
- Controlando y adaptando el número de partículas, se adapta fácilmente a los recursos computacionales disponibles.
- Fácil de implementar desde el punto de vista computacional.

Inconvenientes:

- Si el conjunto de partículas es pequeño, un robot bien localizado puede llegar a perderse debido a que el proceso de remuestreo aleatorio no genere ninguna muestra en la localización correcta.
- Para resolver también el problema de la recuperación ante fallos (kidnapping) es necesario alterar el algoritmo estándar, manteniendo siempre un número reducido de partículas aleatoriamente distribuidas por el entorno.

3.4 Planificación Local

Una vez analizadas diferentes aplicaciones de localización y mapeado del entorno, el siguiente paso será la planificación tanto local como global. Esta sección se dedica a aplicaciones de planificación local. Primero se implementa una evitación de obstáculos partiendo de un algoritmo ejemplo de la *Robotics System Toolbox*, tras el cual se programa un algoritmo propio de seguimiento de pasillos.

3.4.1 Evitación de obstáculos

En este apartado se analizarán los planificadores locales también conocidos como evitadores de obstáculos, con el fin de dotar a la plataforma de una navegación local segura. Para ello se estudiará el planificador local propuesto por Borenstein y conocido como VFH (Vector Field Histogram) [34], que tiene implementaciones tanto en la *Robotics System Toolbox* de Matlab como en ROS.

3.4.1.1 Base teórica: VFH

La evitación de obstáculos es uno de los puntos clave para aplicaciones de robots móviles. Todas incorporan alguna implementación para evitar colisiones, desde algoritmos primitivos que únicamente detectan obstáculos y paran al robot, hasta algoritmos sofisticados que permiten bordear el objeto. Estos últimos son mucho más complejos, ya que no solo implican la detección de obstáculos, sino también alguna medida cuantitativa relacionada con las dimensiones del objeto. Una vez estas han sido determinadas, el algoritmo necesita dirigir al robot alrededor del obstáculo y proceder al destino objetivo original. Normalmente, este procedimiento requiere que el robot se pare frente al obstáculo, obtenga medidas y después reanude el movimiento. La evitación de obstáculos puede dar lugar a caminos no-óptimos, ya que no se usa información del entorno conocido a priori.

VFH es el algoritmo propuesto por Borenstein en el año 1991. Este selecciona la dirección óptima a seguir ante la detección de un obstáculo que no estaba contemplado en el mapa. Usa un histograma cartesiano bidimensional como modelo de entorno, el cual es actualizado continuamente con la información que recoge de los sensores del robot. El método VFH emplea una reducción en dos pasos para calcular los comandos de control deseados para el vehículo. En el primer paso se reducen las casillas de ocupación bidimensionales a un histograma polar de 1D como se muestra en la figura 3.22, construido alrededor de la localización del robot en ese momento. Cada sector del histograma polar contiene un valor que representa la densidad de ocupación en esa dirección. En el segundo paso, el algoritmo selecciona el sector más apropiado de todos con una densidad de ocupación baja y se modifica la dirección del robot acorde a esto.

En el siguiente apartado se explica más detalladamente la implementación de este algoritmo utilizando la *Robotics System Toolbox* de Matlab.

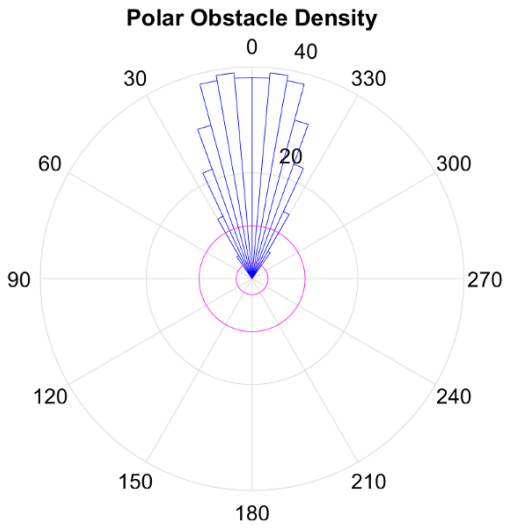

Figura 3.22: Histrograma polar VFH

3.4.1.2 Implementación con Matlab-ROS

La *Robotics System Toolbox* de Matlab proporciona el algoritmo VFH a través del objeto *robotics.VectorFieldHistogram*, además de un conjunto de parámetros que pueden configurar el funcionamiento del mismo.
En esta sección se exponen diferentes pruebas llevadas a cabo para conseguir configurar los parámetros del algoritmo en modo simulado para que, una vez conseguidos los mejores parámetros, se validaran en modo real. Para ello, se empezó estudiando cómo utilizar el evitador de obstáculos.

El objeto *robotics.VectorFieldHistogram* permite al robot evitar obstáculos basándose en datos de los sonar usando el histograma polar VFH. Leyendo datos del láser y teniendo una dirección objetivo, el objeto calcula un camino libre de obstáculos.

Primero, calcula un histograma utilizando los datos del sensor. Convierte todas las direcciones alrededor del robot a sectores angulares y, con las medidas del láser, calcula un histograma de densidad polar sobre estos sectores. Este histograma muestra los sectores angulares en azul y los umbrales en rosa.

Posteriormente, usa los umbrales del histograma para calcular un histograma binario que indica direcciones ocupadas y libres. Finalmente, el algoritmo calcula un histograma enmascarado, el cual es calculado a partir del histograma binario basado en el mínimo radio de giro del robot.

El algoritmo selecciona múltiples direcciones basándose en el espacio y direcciones de conducción posibles. Una función de coste, cuyo peso corresponde a las direcciones previa, actual y objetivo, calcula el coste de las diferentes direcciones posibles. El objeto devuelve una dirección libre de obstáculos con coste mínimo.

Para crear el objeto se usa la sintaxis:
VFH = robotics.VectorFieldHistogram(Name,Value), que devuelve un histograma polar con opciones adicionales especificadas por uno o más pares (Name, Value). *Name* es el nombre de la propiedad y *Value* es su valor correspondiente.

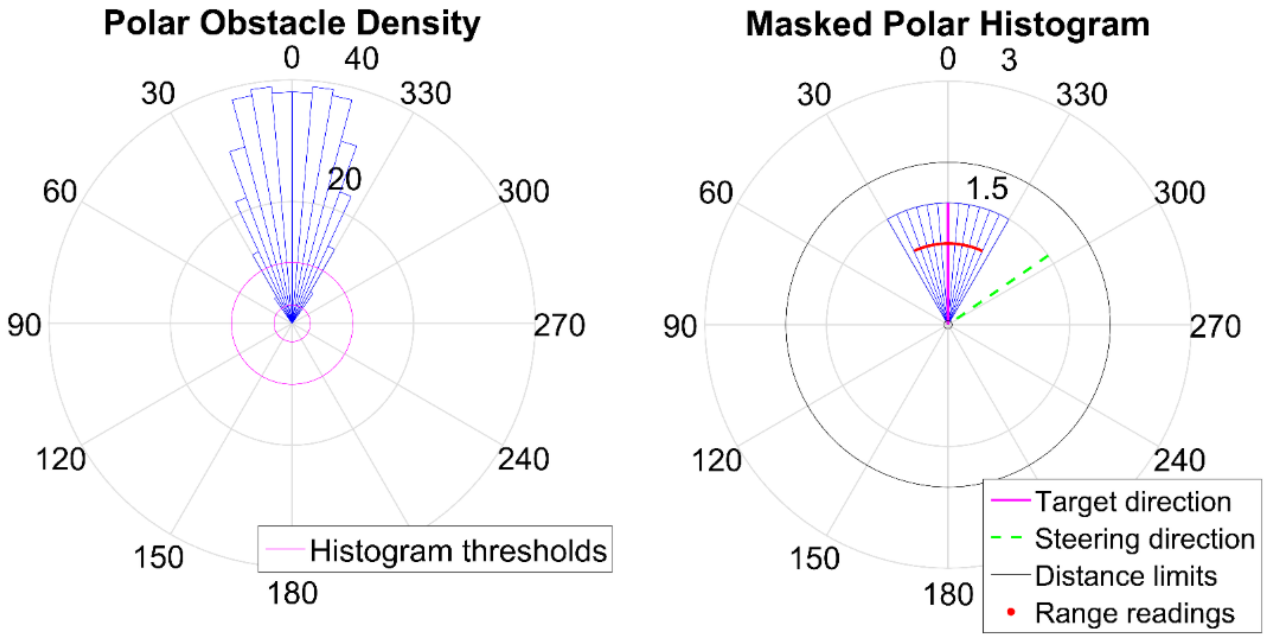

Figura 3.23: VFH: Histograma polar e histograma polar enmascarado.

A continuación se describen las propiedades de este:

- *NumAngularSectors*: Número de sectores angulares.
- *DistanceLimits*: especifica las distancias de interés para considerar la evitación de obstáculos. El límite inferior se usa para ignorar lecturas del láser que intersecten con partes del robot, la poca precisión del sensor a distancias cortas, o ruido del sensor. El límite superior es el rango efectivo del sensor o la máxima distancia que se quiera considerar.
- *RobotRadius*: radio del robot. Este radio asegura que el robot evita obstáculos basándose en su tamaño.
- *SafetyDistance*: distancia de seguridad alrededor del robot.

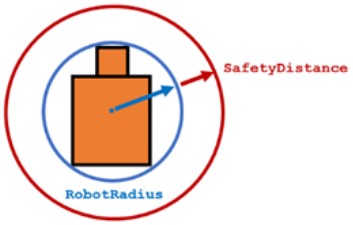

Figura 3.24: SafetyDistance

- *MinTurningRadius*: radio de giro mínimo para el robot desplazándose a la velocidad deseada.

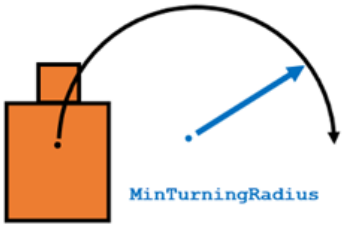

Figura 3.25: MinTurningRadius

- *HistogramThresholds*: [3 10] por defecto. Umbrales para el cálculo del histograma enmascarado. Zonas con valores de densidad mayores que el umbral máximo son representados como espacio ocupado (1) y los que tienen valor menor que el mínimo umbral son representados como espacio libre (0). Los que tienen un valor intermedio son representados con el histograma previo, siendo por defecto espacio libre. Este histograma muestra la dirección actual y destino, las medidas del sensor y la distancia límite. Este puede verse en la figura 3.23.

- *UseLidarScan*: usar un objeto *lidarScan* como entrada.

Los pesos de la función de coste se usan para calcular la dirección final utilizando la dirección actual previa y objetivo. Cambiar estos pesos afecta a la capacidad de respuesta del robot y a la reacción ante posibles obstáculos. Para que el robot se dirija a la dirección objetivo, el peso de *TargetDirectionWeight* ha de ser mayor. Las propiedades que especifican estos pesos son las siguientes:

- *TargetDirectionWeight*: Peso de la función de coste para la dirección objetivo (5 por defecto).

- *PreviousDirectionWeight*: Peso de la función de coste para la dirección previa (2 por defecto).

- *CurrentDirectionWeight*: Peso de la función de coste para la dirección actual (2 por defecto).

steeringDir = vfh(scan,targetDir) encuentra una dirección libre de obstáculos usando el algoritmo VFH+ para la medida del láser *scan* y la dirección objetivo *targetDir*.

En el siguiente apartado se muestra cómo se implementó este algoritmo así como los resultados obtenidos en las diferentes plataformas.

3.4.1.3 Resultados

El punto de partida es el mapa obtenido empleando técnicas de SLAM, en concreto, empleando el algoritmo *gmapping*, al cual se insertó un obstáculo virtual en el mapa real dibujando para ello una caja en el centro del pasillo que obstaculice el paso del robot, como se observa en la figura 3.26.

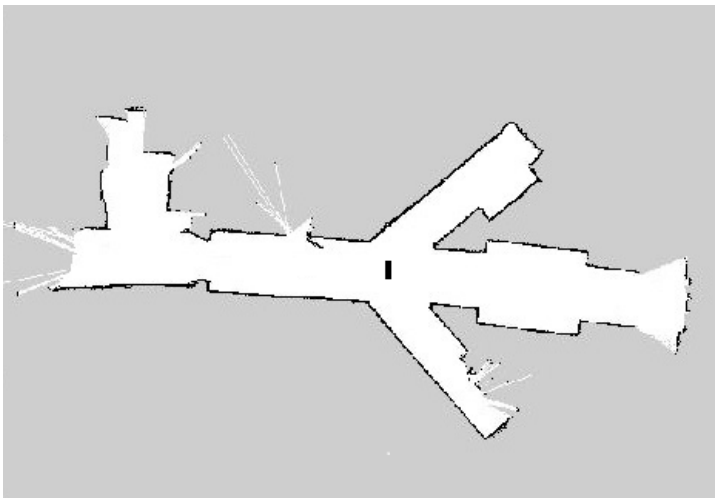

Figura 3.26: Mapa utilizado con el algoritmo VFH.

El algoritmo implementado es el siguiente:

1. Conectar a ROS (simulador/robot real)
2. Crear un suscriptor que reciba los datos del láser (/robot0/laser_1)
3. Crear un publicador para el comando de velocidad (/robot0/cmd_vel)
4. Especificar las propiedades del algoritmo VFH.
5. En bucle:
   a) Recibir datos del láser
   b) Llamar al objeto VFH para obtener la dirección del robot utilizando la función *step*, pasando como parámetros el objeto, los datos del láser (en rangos y ángulos) y la dirección objetivo.
   c) Mostrar por pantalla el histograma con el método *show*.
   d) Calcular velocidades. Si la dirección obtenida es válida calcula la velocidad angular llamando a *exampleHelperComputeAngularVelocity* (pasando como parámetros la dirección del robot y la velocidad angular máxima en rad/s), fijando la velocidad lineal a 0.1 m/s. Si la dirección no es válida, para el robot y fija la velocidad angular a 0.1 m/s^2 para buscar una dirección válida.
6. Enviar comando de velocidad al robot.

Una vez preparado el mapa con el obstáculo, se enfrentó el robot al obstáculo y, empleando el método *show* para depurar el comportamiento, se realizaron pruebas para ajustar los valores de las distintas propiedades explicadas anteriormente a sus valores óptimos para nuestra aplicación:

1. `Ajuste de la variable` *`targetDir`*. Fijándola a 0 (dirección rectilínea) y dejando los parámetros del VFH a los valores por defecto se comprobó que el robot avanza por el pasillo evitando los obstáculos que encuentra a su paso.

2. `Ajuste de la propiedad` *`NumAngularSectors`*. Se comprobó que cuando se deja el valor por defecto (180) el algoritmo presenta fallos en algunas ocasiones. Lo mismo sucede para valores mayores; sin embargo, para valores menores, el algoritmo funciona correctamente evitando todos los obstáculos que encuentra a su paso. Para un valor de 50 el aspecto del VFH es el mostrado en la figura 3.27.

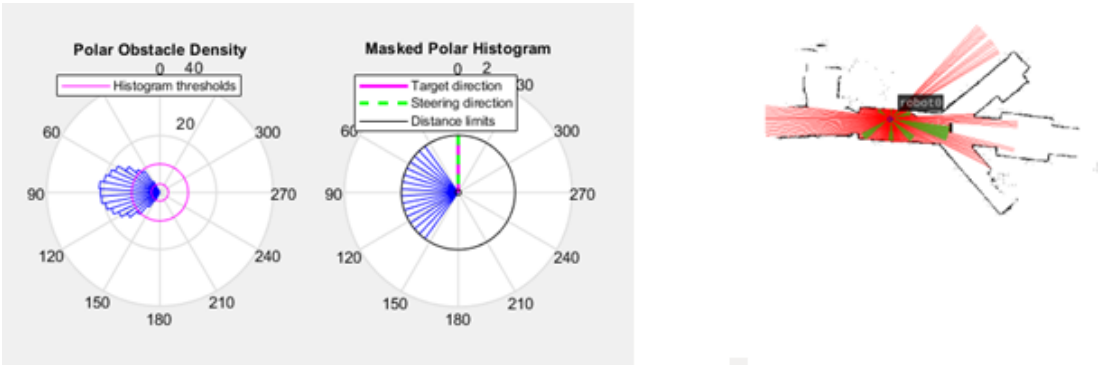

Figura 3.27: Histogramas polares ajustando la propiedad *NumAngularSectors* a 50.

Para un valor de 300 el aspecto del VFH es el mostrado en la figura 3.28.

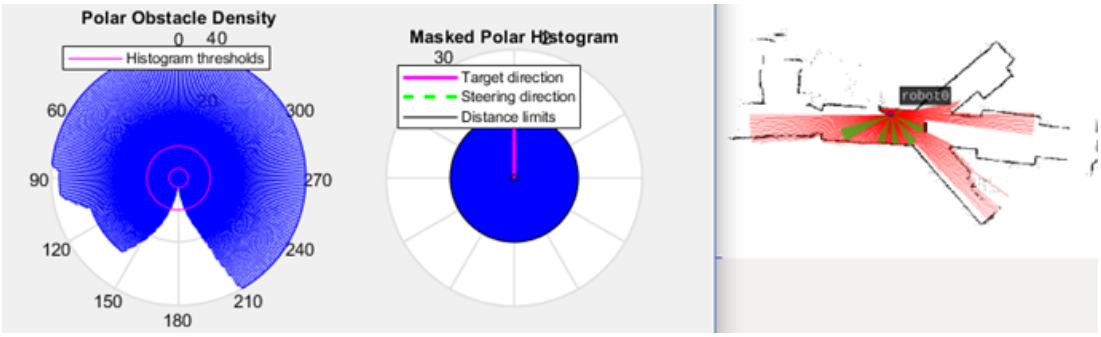

Figura 3.28: Histogramas polares ajustando la propiedad *NumAngularSectors* a 300.

3. `Ajuste de la propiedad` *`DistanceLimits`*. Se comprobó que con un valor muy alto del límite superior se tienen en cuenta obstáculos que están a una gran distancia del robot, lo cual no es necesario. Es preferible mantener un valor que se ajuste al tamaño del robot y del pasillo. El valor por defecto permite detectar obstáculos a un metro de distancia; el límite inferior (0.05m) permite ignorar falsos obstáculos por la precisión del sensor a rangos bajos.

4. `Ajuste del parámetro` *`RobotRadius`*. El valor más adecuado es 0.15m ya que es el radio real del robot. Si se ajusta un valor más bajo es posible que el robot choque con algún obstáculo, ya que el algoritmo considera que tiene un tamaño menor. Si, por el contrario, se ajusta un valor alto, puede que el robot quede atascado en una zona por la que realmente si puede pasar, ya que el algoritmo cree que este tiene un tamaño mayor.

5. `Ajuste del parámetro` *`SafetyDistance`*. Esta es la distancia de seguridad a dejar alrededor del robot a parte de su radio. Se usa la suma de la distancia de seguridad y el radio del robot para encontrar la dirección libre de obstáculo. Lo que ocurre es similar al apartado anterior, ya que la funcionalidad es la misma.

6. `Ajuste del parámetro` *`MinTurningRadius`*. Es el mínimo radio de giro para el robot moviéndose a su velocidad actual. El valor por defecto es 0.1m. Se comprueba, usando un valor de 1m, que un valor tan elevado como este empeora el funcionamiento del algoritmo, ya que cuando encuentra un obstáculo a su paso debe girar como mínimo 1m. Teniendo en cuenta las dimensiones del pasillo y del robot, este valor tan elevado hace muy difícil que el robot consiga avanzar por este.

   Sin embargo, si se usa un valor muy bajo (0.001m por ejemplo) se observa, que cuando el robot encuentra un obstáculo a su paso, gira lo justo y necesario, ya que el parámetro fija el valor mínimo de radio de giro, pero no el máximo. Por tanto, con valores pequeños siempre funcionará bien.

7. `Ajuste del parámetro` *`TargetDirectionWeight`*. Por defecto, *TargetDirectionWeight* tiene un valor de 5, mientras que *CurrentDirectionWeight* y *PreviousDirectionWeight* tienen un valor de 2. Para seguir una dirección hay que usar un valor de *TargetDirectionWeight* mayor que la suma de los otros dos, lo cual ocurre en los valores por defecto. Para ignorar el coste de la dirección hay que fijar el valor a 0, donde se comprueba que el robot no sigue ninguna dirección concreta. Cuando detecta un obstáculo, gira hasta encontrar algún camino libre, quedándose siempre en la misma zona del pasillo. Finalmente, se fijó un valor de 6.

8. `Ajuste del parámetro` *`CurrentDirectionWeight`*. Valores altos de este peso producen caminos eficientes. Se probó con un valor de 10 y se observó que el robot se quedaba parado cuando encontraba el primer obstáculo, como se muestra en la figura 3.29.

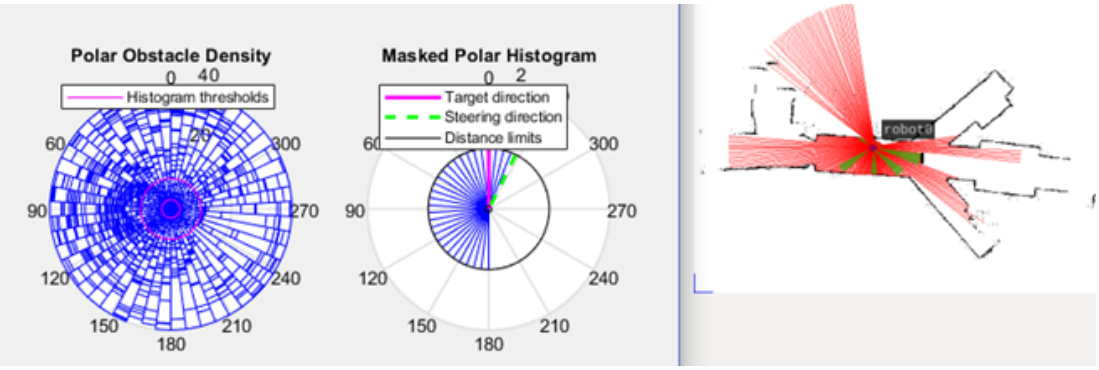

Figura 3.29: Histogramas polares con *CurrentDirectionWeight* = 10.

Si se fija un valor de 0, se ignora el coste de la dirección actual. Haciendo esto se observó el mismo comportamiento que anteriormente. Finalmente se fijó un valor de 2.

9. `Ajuste del parámetro` *`PreviousDirectionWeight`* **.** Valores altos de este peso producen caminos suaves. Realizando el mismo experimento que en el apartado anterior, se comprobó que con un valor de 10 se observaba exactamente el mismo comportamiento. Finalmente se fijó un valor de 2.

10. `Ajuste del parámetro` *`HistogramThresholds`* Se hicieron distintas pruebas y se fijó finalmente a un valor de [0.2 6], valores en los que los objetos son detectados perfectamente.

Tras realizar el ajuste de todos los parámetros y comprobar el correcto funcionamiento en simulación (*VFH_SIMULACION.m*), se pasó a probar el algoritmo sobre las plataformas reales (*VFH_ROBOT.m*). Este es el mismo que el especificado para simulación. Únicamente se modificó el nombre de los topics a los que se subscribe y publica (*/scan* y */cmd_vel*). Además hay que habilitar y activar los motores. Se comprobó que, con aproximadamente los mismos parámetros que los ajustados en simulación, el robot evitaba los obstáculos correctamente, salvo en ciertas ocasiones en las que el obstáculo no era visto por los sensores debido a su altura o color.

3.4.2 Seguimiento de pasillos

Una vez explorada la funcionalidad principal de la planificación local, se procede a la elaboración de un algoritmo propio cuya función es el seguimiento de pasillos. Se busca que el robot siga un pasillo recto por el centro del mismo. Para ello se hace uso de la transformada de Hough para la extracción de líneas y de técnicas de control borroso para el seguimiento del pasillo por el centro del mismo.

3.4.2.1 Base teórica: extracción de líneas y control borroso.

La transformada de Hough [54] [55] es una técnica usada para aislar características de una forma particular dentro de una imagen. Requiere que las características deseadas sean especificadas de forma paramétrica, por ello, tradicionalmente ha sido usado para detectar curvas regulares como líneas, círculos, elipses, etc. Una transformada de Hough generalizada puede ser empleada en aplicaciones donde no es posible una descripción analítica de una característica concreta. La principal ventaja de esta técnica es que tolera huecos o espacios en las curvas y es muy poco afectada por el ruido en la imagen. De hecho, su objetivo es resolver estos problemas, haciendo posible realizar agrupaciones de los puntos que pertenecen a los bordes de posibles figuras a través de un procedimiento de votación sobre un conjunto de figuras parametrizadas, donde el umbral de votos para considerar que una línea existe en la imagen será especificado por el usuario.

El modo de funcionamiento es estadístico y de acuerdo a los puntos que se tengan se debe averiguar las posibles líneas en las que puede estar el punto, lo cual se logra por medio de una operación que es aplicada a cada línea en un rango determinado. La trasformada Hough utiliza dentro de su funcionamiento una representación paramétrica de forma geométrica, es decir que, si se tiene una recta, esta se representaría con los parámetros r y θ, donde r es la distancia entre

la línea y el origen, y θ es el ángulo de la proyección perpendicular desde el origen a la línea medida en el sentido de las agujas del reloj desde el eje x positivo, como se muestra en la figura 3.30. Por medio de la parametrización, la ecuación de la recta se podría escribir como r = x·cosθ + y·sinθ.

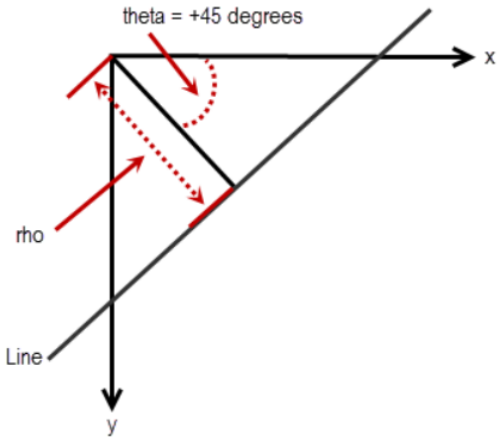

Figura 3.30: Representación de la recta en coordenadas polares.

Como primer paso se puede usar un detector de bordes para obtener los puntos de la imagen que pertenecen a la frontera de la figura deseada. Luego, se transforma cada punto (x,y) en los puntos (r_i, θ_i), obteniendo el espacio de Hough para el conjunto de rectas en dos dimensiones. Para un punto, las rectas que pasan por ese punto son los pares (r, θ) con n r = x·cosθ + y·sinθ. Esto corresponde a una curva sinusoidal en el espacio (r,θ) que es única para ese punto. Si las curvas correspondientes a dos puntos se interceptan, el punto de intersección en el espacio de Hough corresponde a una línea en el espacio de la imagen que pasa por estos dos puntos. Así, un conjunto de puntos de una recta producirán sinusoides que se interceptan en los parámetros de esa línea.

El algoritmo usa un acumulador, matriz de dimensión igual al numero de parámetros desconocidos. En el caso de una recta, la dimensión será dos, correspondientes a los valores cuantificados para (r,θ). Cada celda del acumulador representa una figura cuyos parámetros se pueden obtener a partir de la posición de la celda. Cada punto en la imagen vota por las posibles rectas a las que puede pertenecer ese punto, buscando todas las posibles combinaciones de valores para parámetros que describen la figura, obtenidos a partir del acumulador. Las figuras se pueden detectar buscando las posiciones del acumulador con mayor valor, para lo que se suele usar un umbral.

Con el fin de aclarar la idea expuesta, se muestra un pseudocódigo de como debería funcionar el algoritmo para detectar rectas en una imagen:

1: Cargar imagen

2: Detectar los bordes en la imagen

3: Por cada punto en la imagen:

A: Si el punto (x,y) esta en un borde:

a: Por todos los posibles ángulos θ

i: Calcular ρ para el punto (x,y) con un ángulo θ

ii: Incrementar la posición (ρ,θ) en el acumulador

4: Buscar las posiciones con los mayores valores en el acumulador

5: Devolver las rectas cuyos valores son los mayores en el acumulador.

La figura 3.31 muestra los resultados de la transformada de Hough en una imagen que contiene dos líneas bien definidas:

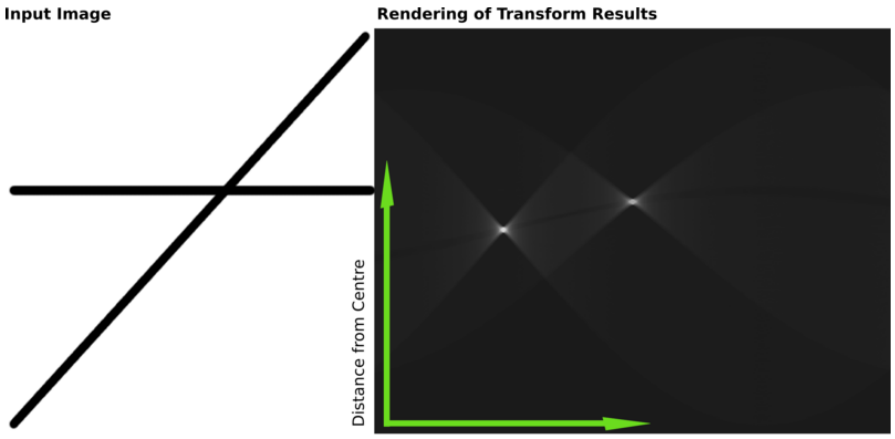

Figura 3.31: Resultados transformada de Hough.

Los resultados de esta transformación son almacenados en una matriz. Los valores de cada celda representan la cantidad de curvas que pasan por un punto. Cuanto mayor sea el valor de la celda más claro se verá. Los dos puntos más claros de la imagen representan las dos líneas.

Una vez explicada la transformada de Hough, la cual se usará para la extracción de líneas e identificación de pasillos, se procede a explicar el concepto de lógica borrosa. Se consideró que el uso de este tipo de control garantizaría un control más preciso del robot, de manera que este siguiese el pasillo guardando la misma distancia a un lado y a otro.

La teoría de Conjuntos Borrosos [56] [57] se basa en el reconocimiento de que determinados conjuntos poseen unos límites imprecisos. Estos conjuntos están constituidos por colecciones de objetos para los cuales la transición de pertenecer o no a los mismos no es abrupta, sino gradual.

Supóngase, por ejemplo, que se desea evaluar la temperatura de una habitación. El universo de discusión lo constituyen todas las posibles temperaturas a las que puede encontrarse la misma. Si se pretende clasificar dichas temperaturas en varios conjuntos, como muy *alta*, *alta*, *media*, *baja* y *muy baja*, pueden seguirse dos alternativas:

- Utilizar Conjuntos Clásicos, de forma que los límites de dichos conjuntos están perfectamente definidos (por ejemplo, la temperatura es *media* si es mayor que 20 grados y menor que 25). Los límites de estos conjuntos están claramente delimitados y todos los elementos de un conjunto pertenecen al mismo por igual. La pertenencia de una temperatura a un conjunto se evalúa de forma binaria : o pertenece (1) o no pertenece (0).

- Utilizar Conjuntos Borrosos, de forma que una determinada temperatura pueda pertenecer a diferentes conjuntos en grados distintos. De esta forma, la transición de un conjunto a otro no es abrupta, sino gradual. La pertenencia de una temperatura a un conjunto no es *todo o nada*, sino que viene determinada por un número real en el intervalo (0,1).

Este tipo de lógica toma dos valores aleatorios, pero contextualizados y referidos entre sí. Es una aproximación a la codificación basada en grados de verdad, en lugar de verdadero o falso como hace la lógica booleana o clásica (0,1), lo cual es más parecido al lenguaje natural. La lógica borrosa incluye 0 y 1 como casos extremos, pero también incluye términos intermedios. A este grado de verdad se le llama grado de pertenencia o posibilidad. El proceso que sigue la lógica borrosa o *fuzzificación* consiste en determinar la posibilidad de pertenencia a un conjunto difuso, el cual no es más que un conjunto de observaciones sobre las que se aplica la misma función de pertenencia. Para cada conjunto difuso, existe asociada una función de pertenencia para sus elementos, que indica en qué medida el elemento forma parte de ese conjunto difuso. Así, un conjunto difuso proporciona una transición suave entre los límites de lo que sería un conjunto discreto o *crisp*. El *Universo del discurso* se define como todos los posibles valores que puede tomar una determinada variable.

En la teoría de conjuntos difusos se definen también las operaciones de unión, intersección, diferencia, negación o complemento, y otras operaciones sobre conjuntos, en los que se basa esta lógica.

Se basa en un sistema de inferencia. Este sigue un procedimiento que consiste en combinar los grados de verdad de distintas variables observadas y tomar una decisión en base a ellas con un razonamiento lógico. Combina dos premisas o dos números difusos para obtener una conclusión de la forma a · b ->c, esta a su vez puede ser un conjunto difuso o un valor discreto.
Los métodos de inferencia para esta base de reglas deben ser sencillos, versátiles y eficientes. Los resultados de dichos métodos son un área final, fruto de un conjunto de áreas solapadas entre sí (cada área es resultado de una regla de inferencia). Para escoger una salida concreta a partir de tanta premisa difusa, el método más usado es el del centroide, en el que la salida final será el centro de gravedad del área total resultante.
Las reglas de las que dispone el motor de inferencia de un sistema difuso pueden ser formuladas por expertos o aprendidas por el propio sistema, haciendo uso en este caso de redes neuronales para fortalecer las futuras tomas de decisiones.

Ahora se explicará la inferencia de Mamdani, el método implementado en este trabajo. Éste es probablemente el más utilizado, propuesto por Ebrahim Mamdani en 1975 [58]. Consiste fundamentalmente en cuatro pasos:

1. `Fuzzificación de las variables de entrada`: consiste en tomar los valores discretos de las entradas y determinar el grado de pertenencia de estas a los conjuntos difusos asociados.

2. `Evaluación de las reglas`: Tomamos las entradas anteriores y se aplican a los antecedentes de las reglas difusas. Si una regla tiene múltiples antecedentes, se utiliza el operador

AND u OR para obtener un único número que represente el resultado de la evaluación. Este número se aplica al consecuente, aplicando un recorte o escalado según el valor de verdad del antecedente. El método más comúnmente utilizado es el recorte, que corta el consecuente con el valor de verdad del antecedente. El escalado proporciona un valor más preciso, preservando la forma original del conjunto difuso. Se obtiene multiplicando todos los valores por el valor de verdad del antecedente.

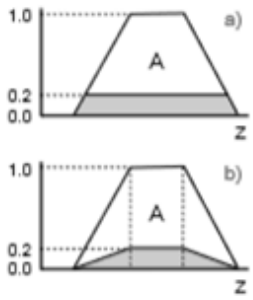

Figura 3.32: Conjunto recortado (a) y escalado (b).

3. `Agregación de las salidas de las reglas`: La agregación es el proceso de unificación de las salidas de todas las reglas; es decir, se combinan las funciones de pertenencia de todos los consecuentes previamente recortados o escalados, combinando para obtener un único conjunto difuso por cada variable de salida.

4. `Defuzzificación`: El resultado final habitualmente es necesario expresarlo mediante un valor discreto. En esta etapa se toma como entrada el conjunto difuso anteriormente obtenido para dar un valor de salida. Existen varios métodos de defuzzificación, pero probablemente el más ampliamente usado es el centroide, como adelantamos anteriormente.

Habiendo explicado la base teórica de las herramientas usadas para la programación del algoritmo de seguimiento de pasillos, se procede a mostrar los pasos para la implementación del mismo en un entorno Matlab/Simulink-ROS aplicando la teoría expuesta en este apartado.

3.4.2.2 Implementación con Matlab-ROS

En Matlab (*SEGUIMIENTO_PASILLO.m*), tras conectarse a ROS creando los correspondientes subscriptores (láser y odometría) y publicadores (comando de velocidad y habilitación de motores), se implementa un bucle periódico. El algoritmo seguido es el siguiente:

1. Se recogen los datos del láser y se transforman a una imagen binaria. Para ello, primero se pasa a coordenadas cartesianas leyendo los datos del láser, scanMsg, con la función *readCartesian*. Hay que realizar un ajuste para que la imagen salga bien orientada.

```
ranges = double(scanMsg.Ranges);
angles = double(scanMsg.readScanAngles);
```

```
odompose = odom.LatestMessage;

cart = readCartesian(scanMsg);
cart = [-cart(:,2) -cart(:,1)];

hold on;
x = cart(:,1);
y = cart(:,2);
```

2. Con estas coordenadas y haciendo uso de *histc* y *accumarray* se convierte a imagen binaria:

```
minx = -8;
maxx = 8;
miny = -8;
maxy = 8;

target_rows = 400;
target_cols = 400;
xsteps = linspace(minx, maxx, target_cols+1);
xsteps(end) = inf;
ysteps = linspace(miny, maxy, target_rows+1);
ysteps(end) = inf;
[~, xbin] = histc(x, xsteps);
[~, ybin] = histc(y, ysteps);

BW = accumarray([ybin, xbin], 1, [target_rows, target_cols]);
bwimg = BW > 0;
imshow(BW); title('scan')
```

3. Una vez hecho esto, se aplica la transformada de Hough para encontrar las posibles líneas en dicha imagen. Se obtienen un conjunto de líneas con sus correspondientes puntos inicial y final, y sus valores de *theta* y *rho*. Se realiza una corrección de los valores de *theta* y *rho*, ya que debido al sistema de referencia que usa la transformada de Hough (figura 3.30), el valor de *rho* será negativo cuando debería ser positivo según nuestro sistema.

   Para ello, cuando rho sea negativo, se convierte en positivo y se suma 180 grados a theta, acotándolo entre $\pm\pi$ con la función *angdiff*.

4. A continuación, se debe crear un algoritmo que, usando los valores proporcionados por la transformada de Hough, detecte dos líneas paralelas, y corrija la orientación y posición del robot de manera que este se mueva en una trayectoria paralela a estas líneas y por el centro del pasillo.

Lo primero que se debe hacer es cambiar el marco de referencia que proporciona la transformada de Hough al marco de referencia del robot. Para ello se crea la función *cambioArobot.m.* Se introducen como parámetros de entrada el ángulo *theta* y la distancia *rho*, los cuales definen un vector perpendicular a la recta en cuestión y se obtiene el punto final que define este vector en el marco de coordenadas del robot.

Para ello, mediante una matriz de transformación homogénea se realiza una traslación de 200 celdas en eje x e y, una rotación respecto al eje z del sistema inicial (móvil) de -90 grados y otra respecto al eje x del sistema móvil de 180 grados. Como se obtiene mediante rotaciones y traslaciones definidas con respecto al sistema móvil, la matriz homogénea que representa cada transformación se deberá postmultiplicar sobre las matrices de las transformaciones previas. De esta manera conseguimos situarnos en el marco de referencia del robot. Con esta matriz de transformación homogénea podemos transformar cualquier vector dado en uno de los sistemas al otro. Así, transformamos el punto definido por *theta* y *rho* en el sistema original P_m al sistema de referencia del robot P_f.

```
Tfm=transl(200,200,0)*trotz(-pi/2)*trotx(pi);
Pm=[ro*cos(theta); ro*sin(theta); 0; 1];
Pf=Tfm*Pm;
```

5. Con este punto y la pendiente (`m = tan(pi-theta)`), se obtiene la ecuación de la recta y con esta el punto de corte con el eje y, que será la distancia de la recta al robot. Esta distancia ha de ser multiplicada por el factor que relaciona metros y celdas.

6. Por geometría se halla también el ángulo que forma dicha recta con el eje x, que será la orientación que deba tomar el robot para ir paralelo a esta (`-theta`). Para asegurarnos de que el ángulo que escoge es el del sentido de avance del robot, debemos asegurarnos que el valor absoluto de este es menor que $\pi/2$. En caso contrario, se substrae este valor de π y se limita entre $\pm\pi$.

```
x0=Pf(1);
y0=Pf(2);

 %Pendiente de la recta
m=tan(pi-theta);

 %Ángulo de la recta a seguir
angulo=-theta;

if abs(angulo)>pi/2  % Tomamos el sentido de avance del robot
  angulo=angdiff(pi,angulo);
end

 %Corte con el eje y
```

```
d=y0-(m*x0);

d=d*k;
```

7. El siguiente paso es obtener la distancia del robot a la línea central del pasillo. Para ello, de todas las líneas captadas por la transformada de Hough, debemos escoger aquellas que definan mejor las paredes del pasillo. En la primera iteración se realiza una media de todos los ángulos obtenidos con la función *cambioArobot.m* y se cogen aquellas dos líneas dadas por la transformada de Hough cuya orientación sea más similar a esta media. Para las siguientes iteraciones, se cogen aquellas dos líneas cuya orientación sea más similar a la anterior, discriminando aquellas líneas cuya separación sea menor que 0.1m (misma pared). Usando el corte con el eje y correspondiente a estas dos líneas, se calcula la distancia del robot al centro del pasillo:

```
dist = (abs(d1)+abs(d2))/2; %distancia de la pared al centro
if d1>0
  dpos=d1;
else
  dpos=d2;
end
distance = dpos - dist; %distancia del robot al centro
```

8. Usando como parámetros de entrada la distancia al centro del pasillo y el ángulo del robot respecto a la dirección paralela al pasillo se aplica lógica difusa, cuya salida será la velocidad angular óptima. Para ello hacemos uso de la aplicación de Matlab *Fuzzy Logic Designer*. Concretamente diseñamos un sistema de inferencia borrosa de tipo Mamdani (*fuzzy_mamdani.fis*). El primer paso es introducir las variables de entrada y de salida especificadas y escribir las reglas que deberá usar el controlador, mostradas en la figura 3.33.

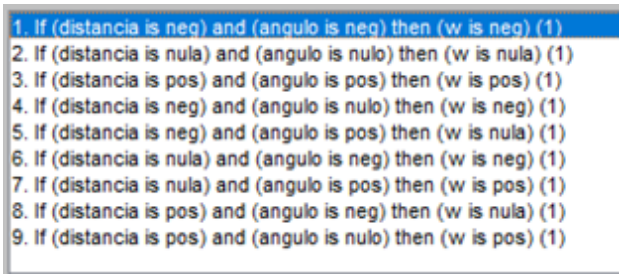

Figura 3.33: Reglas del controlador borroso.

Los números entre paréntesis representan los pesos de cada regla, que para nosotros será el mismo para todas ellas.

9. Tras esto se hace uso del *Membership Function Editor* para editar y representar todas las funciones de pertenencia asociadas a cada una de las variables de entrada y salida del sistema. En nuestro caso, cada una de las variables tendrá tres funciones de pertenencia. A cada una

de ellas se le asociará un nombre y unos parámetros: media y desviación estándar de la curva gaussiana. Así mismo se indica el rango de cada una de las variables. A continuación se muestran los gráficos de las funciones de pertenencia de cada una de las variables:

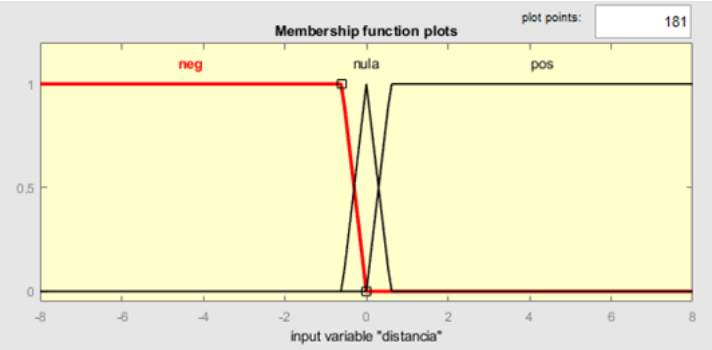

Figura 3.34: Función de pertenencia de la variable distancia.

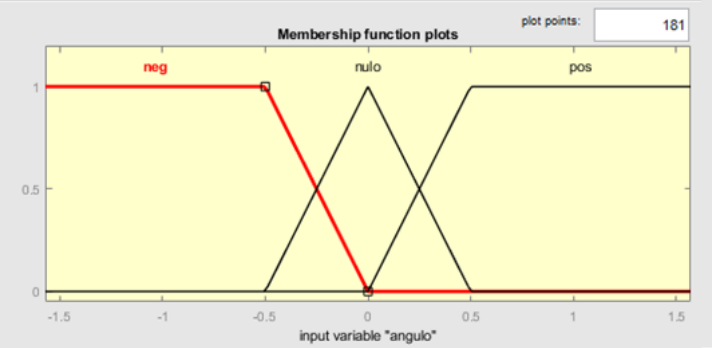

Figura 3.35: Función de pertenencia de la variable ángulo.

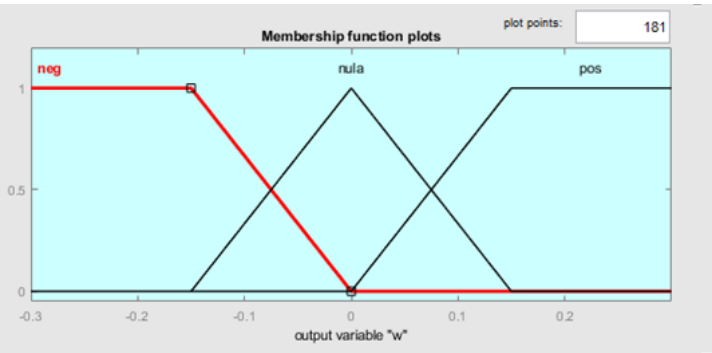

Figura 3.36: Función de pertenencia de la variable w.

En el *Rule Viewer* (figura 3.37), podemos ver los gráficos que representan el proceso de inferencia según el valor de las entradas. Las dos primeras columnas muestran las funciones de pertenencia referenciadas por el antecedente, o la parte *if* de cada regla. La tercera columna muestra las funciones de pertenencia referenciadas por el consecuente o la parte *then* de cada regla. El último gráfico en la tercera columna representa la decisión final del sistema de inferencia. La salida discreta o *defuzzified* es mostrada con la línea roja vertical.

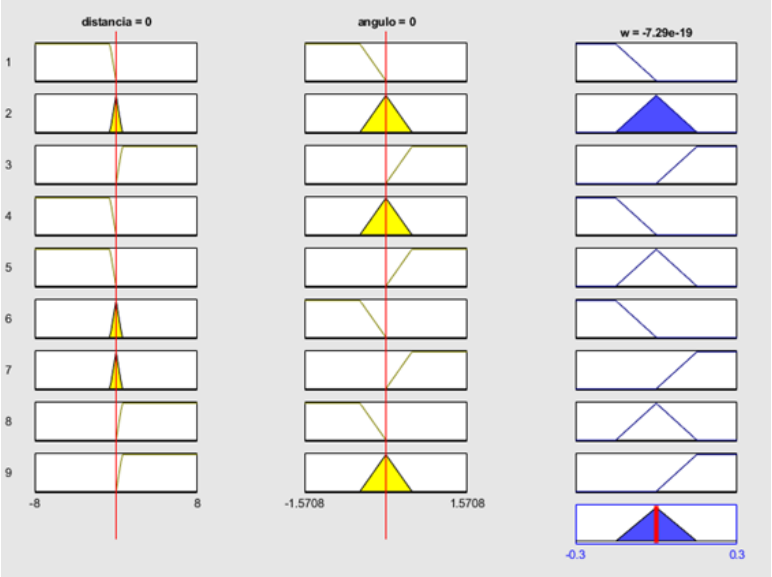

Figura 3.37: Rule Viewer.

10. Por último, la velocidad angular proporcionada por el controlador es publicada al topic cmd_vel del robot o simulador.

Una vez explicado el procedimiento seguido para la elaboración del algoritmo de seguimiento de pasillos, se exponen en el siguiente apartado los resultados obtenidos de este en las diferentes plataformas.

3.4.2.3 Resultados

La respuesta obtenida en simulación y con el robot Amigobot es muy similar. El algoritmo responde satisfactoriamente aunque presenta oscilaciones cuando se encuentra amplios espacios abiertos en el pasillo. El código contempla la opción de posibles discontinuidades pero, dado que se trata de unas discontinuidades muy grandes y a ambos lados del robot, este pierde ligeramente su trayectoria al pasar por esta zona. Aun así, recupera rápidamente el control al pasar estos espacios abiertos. Esto puede observarse en el vídeo que se aporta como ejemplo de ejecución de esta aplicación sobre la plataforma Amigobot. El enlace para acceder a este vídeo se encuentra en la sección C.3.

En la figura 3.38 se observa uno de los gráficos que muestra las líneas de Hough identificadas en el entorno a través de las medidas recibidas del láser.

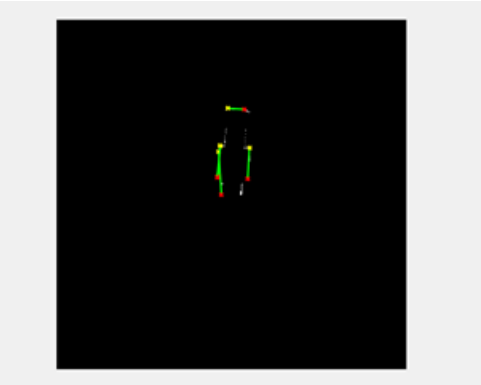

Figura 3.38: Líneas de Hough

3.5 Planificación global

Una vez que se dispone de un navegador local seguro, se propone emplear el navegador global propuesto por Kavraki y conocido como PRM (Probabilistic RoadMap) [60] disponible en la *Robotics System Toolbox* de Matlab, haciendo uso en paralelo del VFH de tal forma que se alcance un destino global garantizando la integridad tanto del robot como del entorno que le rodea.

3.5.1 Base teórica: PRM

El algoritmo de planificación PRM (Probabilistic roadmap method) se aplica al problema de planificación en entornos limitados, creando un camino entre una configuración inicial del robot y una configuración final a la vez que evita colisiones. Principalmente, se basa en coger muestras aleatorias del entorno del robot, categorizarlas como zonas libres u ocupadas y usar un planificador local para conectarlas. Se agregan las configuraciones de inicio y objetivo y se aplica un algoritmo de búsqueda gráfica al gráfico resultante para determinar una ruta entre ambas configuraciones.

El algoritmo PRM consiste en dos fases: una de construcción y otra de búsqueda. En la fase de construcción se construye el mapa, aproximando los movimientos que se pueden hacer en el entorno. Primero, se crea una configuración aleatoria. Después se conecta a algunos vecinos o configuraciones cercanas, típicamente a los k nodos más cercanos o a todos los que se encuentran a una distancia menor de una distancia predeterminada. Se siguen añadiendo nodos y conexiones hasta que el mapa es suficientemente denso. En la fase de búsqueda, los nodos inicial y final son conectados al gráfico y se obtiene el camino.

Dadas unas condiciones relativamente débiles en la forma del espacio libre, PRM es probabilísticamente completo, lo que quiere decir que según aumenta el número de puntos muestreados, la probabilidad de que el algoritmo no encuentre una ruta se aproxima a cero. La tasa de convergencia depende de las propiedades de visibilidad del espacio libre, donde la visibilidad es determinada por el planificador local. A grandes rasgos, si cada punto puede ver una gran fracción del espacio, y también si una gran fracción de cada subconjunto del espacio puede ver una gran fracción de su complemento, entonces el planificador encontrará rápidamente un camino.

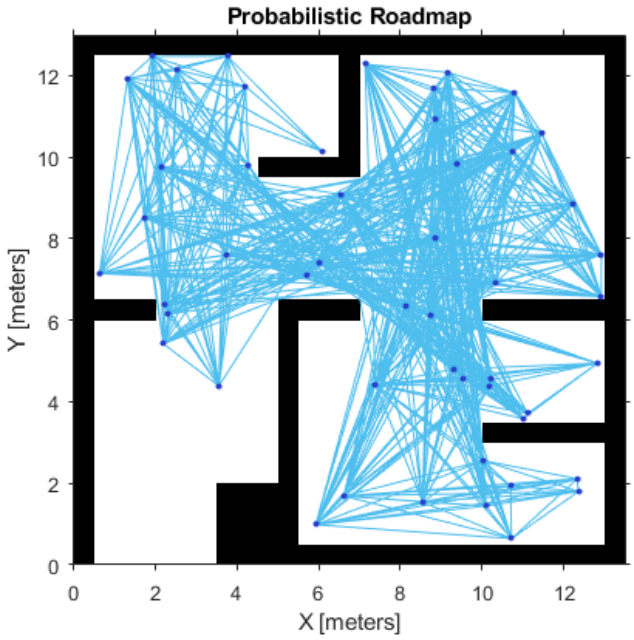

Figura 3.39: Probabilistic Roadmap.

En la siguiente sección se detallan los pasos seguidos para implementar dicho algoritmo empleando el mismo entorno que para el algoritmo de localización (AMCL).

3.5.2 Implementación con Matlab-ROS

Primeramente se realizó un estudio del planificador PRM y se llevaron a cabo una serie de pruebas en simulación para ajustarlo y poder validarlo finalmente en modo real.

La clase *robotics.PRM* genera nodos de manera aleatoria y crea conexiones entre dichos nodos basándose en los parámetros del algoritmo PRM. Los nodos son conectados según las localizaciones de los objetos especificadas en *Map*, y de la distancia especificada en *ConnectionDistance*. Se puede ajustar el número de nodos (*NumNodes*), para especificar la complejidad del mapa y el deseo de encontrar el camino más eficiente. Como se ha explicado en el apartado anterior, el algoritmo PRM usa la red de nodos conectados para encontrar un camino libre de obstáculos desde un principio hasta un final.

A continuación se exponen los pasos seguidos para la elaboración del algoritmo de planificación global con PRM:

1. Se crea el objeto PRM y se ajusta la propiedad *NumNodes*, que especifica el número de nodos en el mapa, los cuales son usados por el algoritmo para generar el mapa de caminos. Cuanto más alto sea este valor, más alta será la complejidad y el tiempo de computación del planificador, pero el camino será más eficiente ya que hay más posibilidades. Para conseguir una buena cobertura del mapa, puede que se necesite un gran número de nodos. Además hay que tener en cuenta que, debido a la colocación aleatoria de los nodos, en algunas áreas del mapa puede que no haya un número suficiente de nodos para conectarlos al resto del mapa.

2. Después se ajusta la propiedad *ConnectionDistance*, la cual indica el umbral superior para los puntos del mapa que están conectados. Cada nodo es conectado con todos los demás nodos dentro de esta distancia siempre que no haya un obstáculo entre ellos. Disminuyendo la distancia de conexión, se puede limitar el número de conexiones para reducir el tiempo de computación y simplificar el mapa. Sin embargo, una distancia baja limita el número de caminos posibles. Cuando se trabaja con mapas simples, puede usarse una distancia de conexión alta con un número pequeño de nodos para incrementar la eficiencia. Para mapas complejos con muchos obstáculos, un alto número de nodos con una distancia de conexión pequeña incrementa las posibilidades de encontrar una solución.

3. Para crear el mapa de caminos hay que llamar a *prm = robotics.PRM(map,__)* o especificar la propiedad *Map* en el objeto PRM.

4. Tras esto, se llama al método show. En este punto los nodos son generados aleatoriamente y se efectúan las conexiones.

   Cuando las propiedades cambian, cualquier método (*update, findpath, o show*) llamado en el objeto activa el nuevo cálculo de los puntos del mapa de caminos y sus conexiones. Como este nuevo cálculo puede ser computacionalmente intensivo, se puede reutilizar el mismo mapa de caminos llamando a *findpath* con diferentes puntos de inicio y final.

Empleando el mismo mapa que en el apartado de planificación local (figura 3.19), y el controlador *Pure pursuit* [60], se proporciona al planificador 4 destinos diferentes dentro del mapa.

Pure pursuit se trata de un algoritmo de planificación que calcula la velocidad angular que moverá al robot de su posición actual a algún punto enfrente suya. La velocidad lineal se asume constante, por tanto, se puede cambiar en cualquier punto. El algoritmo mueve el siguiente punto en la trayectoria basándose en la posición actual hasta llegar al último punto. Se puede pensar como si el robot estuviese persiguiendo siempre un punto en frente suya. La propiedad *LookAheadDistance* decide cómo de lejos se sitúa el siguiente punto.

La clase *robotics.PurePursuit* actúa como un algoritmo de seguimiento para seguir trayectorias. Las velocidades máximas pueden ser especificadas, determinadas por las especificaciones del robot. Dada la posición y orientación del robot como entrada, el objeto calcula los comandos de velocidad lineal y angular. Cómo use el robot estos comandos depende del sistema que se esté usando.

Es importante entender el marco de coordenadas usado por este algoritmo para sus entradas y salidas. Las entradas son coordenadas [x y], que son usadas para calcular los comandos de velocidad del robot. La posición del robot viene dada en [x y θ]. El valor θ es la orientación angular del robot medida en el sentido contrario a las agujas del reloj desde el eje x, como se muestra en la figura 3.40.

La propiedad *LookAheadDistance* es la propiedad principal de este controlador. Determina cómo de lejos debe mirar el robot en el camino desde su posición actual para calcular los comandos de velocidad angular. En la figura 3.41 se observa que el camino actual no coincide con la línea directa entre puntos del camino.

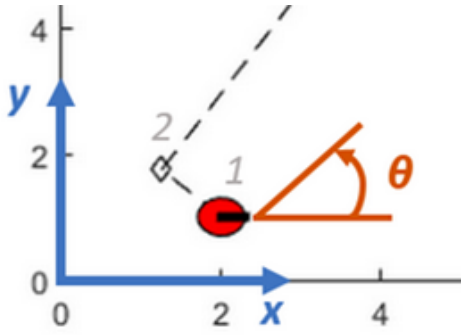

Figura 3.40: Marco de coordenadas de la clase PurePursuit.

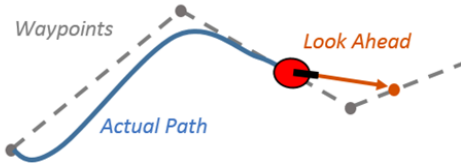

Figura 3.41: LookAheadDistance

La trayectoria depende enormemente de este parámetro y hay dos objetivos principales: recuperar y mantener la trayectoria. Para recuperar rápido el camino entre puntos, es necesario un valor pequeño de *LookAheadDistance*. Sin embargo, como se puede ver en la figura 3.42, el robot lo sobrepasa y oscila a lo largo del camino. Un valor más alto puede dar como resultado grandes curvas cerca de las esquinas. Por tanto, se trata de una situación de compromiso en el que el valor idóneo debe hallarse por experimentación.

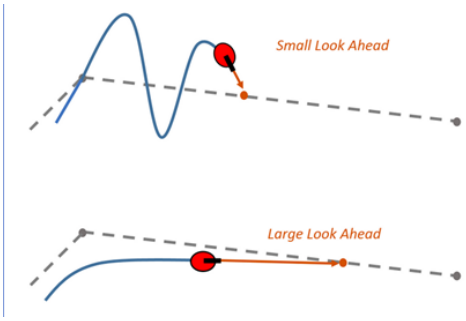

Figura 3.42: Trayectoria según la dimensión del parámetro LookAheadDistance

Con lo expuesto pueden diferenciarse claramente dos limitaciones para el controlador *Pure Pursuit*:

a. No puede seguir exactamente el camino directo entre puntos.

b. Este algoritmo no estabiliza el robot en un punto. Debe ser aplicado un umbral de distancia para parar el robot cerca del objetivo deseado.

En el siguiente apartado se exponen los resultados obtenidos con esta aplicación.

3.5.3 Resultados

En *PathFollowingControllerSimulation.m* se hizo una primera prueba del algoritmo PRM. Para ello, nos suscribimos al topic */robot0/odom* con el fin de recibir la posición y orientación del robot y creamos un Publisher para publicar las velocidades en el topic */robot0/cmd_vel*.

Se indican las posiciones actual y destino y se crea el controlador. Para ello hay que llamar al objeto *robotics.PurePursuit*, definir los puntos del camino a seguir en la propiedad *Waypoints*, la velocidad lineal (0.2 m/s), la máxima velocidad angular (0.3 rad/s) y la propiedad *LookAheadDistance* (0.5 m). Tras diversas pruebas se comprobó que para unos valores de 0.2 m/s, 0.3 rad/s y 0.5m respectivamente, el algoritmo funciona correctamente en simulación. Así mismo se especificó un radio para la posición objetivo de 0.1m, dentro del cual el algoritmo interpreta que ha llegado a su destino.

Para probar el algoritmo PRM junto con el controlador, se llama al objeto *robotics.PRM* pasándole como parámetro de entrada el mapa del pasillo. Este mapa ha de ser modificado previamente, de manera que las paredes y demás obstáculos estáticos aparezcan con una anchura mayor, en nuestro caso de 0.15m (radio del robot), de esta forma los nodos de la trayectoria no pasen cerca de los obstáculos a una distancia menor del radio del robot (figura 3.43). Tras esto se configura el número de nodos y la propiedad *ConnectionDistance*. Tras varias pruebas se observó que con un menor número de nodos respondía mejor. Con 50 nodos y una distancia de 5m el algoritmo responde satisfactoriamente. Llamando a la función *findpath* pasándole como parámetros de entrada el objeto prm y las posiciones inicial y final, el algoritmo crea una trayectoria para el robot. Esta trayectoria ha de ser especificada en la propiedad del controlador *Waypoints*.

Dentro de un bucle que se ejecuta a 10Hz se llama al controlador hasta que el robot se encuentre dentro del radio de la posición de destino. Este calcula las velocidades adecuadas en función de la posición leída de la odometría y el camino proporcionado por el objeto PRM, publicándolas posteriormente en el topic */robot0/cmd_vel*.

```
while( distanceToGoal > goalRadius )

% Salidas del controlador
pos=odom.LatestMessage.Pose.Pose.Position;
initoriq=odom.LatestMessage.Pose.Pose.Orientation;
initori = quat2eul ([initoriq.W initoriq.X initoriq.Y initoriq.Z]); %
   Primero escalar y te devuelve [yaw, p, r]
[v, omega] = controller([pos.X pos.Y initori(1)]);
velMsg.Linear.X = v;
velMsg.Angular.Z = omega;
velPub.send(velMsg);

% Información de la localización del robot
```

```
pos=odom.LatestMessage.Pose.Pose.Position;
robotCurrentPose = [pos.X pos.Y];

% Distancia al destino
distanceToGoal = norm(robotCurrentPose(1:2) - robotGoal)

waitfor(controlRate);

end
```

Se observa que este algoritmo es bastante robusto en simulación. En la figura 3.43 se muestra el camino resultado así como la conexión de los distintos nodos en el mapa y los waypoints generados por el objeto *robotics.PurePursuit*.

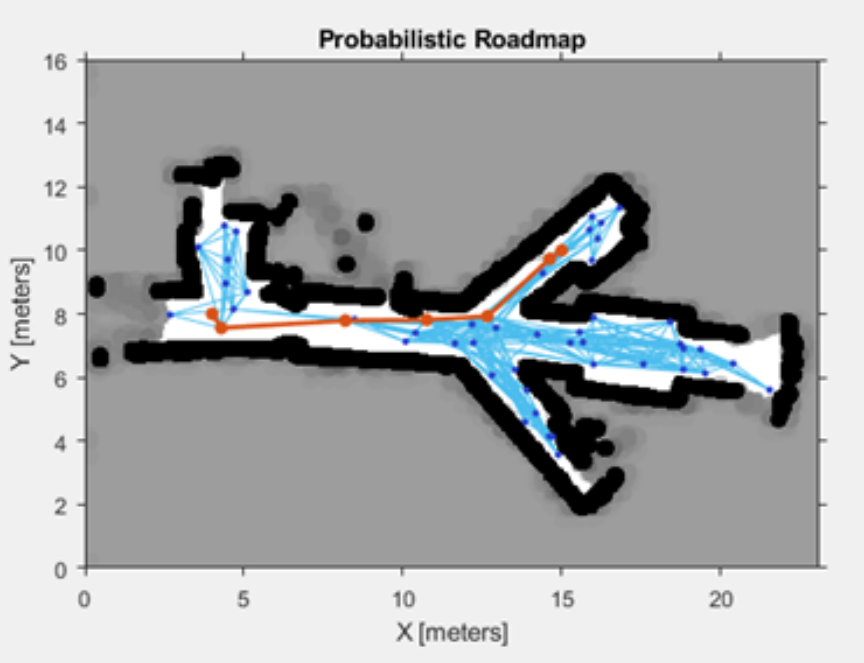

Figura 3.43: Probabilistic Roadmap en simulación.

Tras comprobar su correcto funcionamiento en simulación se procede a probarlo con los robots reales. Únicamente se tuvo que cambiar los topics a los que se suscribe y en los que publica, siendo estos */pose* y */cmd_vel* respectivamente para ambos robots.

Se observa una gran diferencia entre el comportamiento del Amigobot y el Seekur. Aunque de manera mucho menos robusta, el Amigobot presenta una respuesta bastante satisfactoria. Sin embargo, tras muchas pruebas, no se consiguió encontrar una combinación de parámetros óptima para el Seekur, el cual presenta un comportamiento mucho más aleatorio. Por tanto, podemos determinar que este algoritmo no es eficaz para trabajar con el Seekur y tampoco es demasiado robusto con el Amigobot, ya que falla en numerosas ocasiones.

3.6 Integración y resultados globales

Una vez elaborados y ajustados todos los algoritmos expuestos se crea una aplicación global de navegación que una localización y planificación para ver una aplicación real completa.

Para ello se unen los algoritmos VFH, AMCL y PRM en el script de nombre VFH_AMCL_PRM.m. Con esta aplicación el robot va recorriendo el mapa evitando obstáculos (VFH) hasta que se localiza con una covarianza pequeña, menor que 0.05 (AMCL) y, usando la posición estimada proporcionada por este algoritmo, se hace uso del PRM, de manera que cree una ruta hasta el destino y llegue a este mediante un controlador *Pure Pursuit*.

Realizando una serie de pruebas en simulación, se comprobó que primero se localizaba correctamente en el mapa (AMCL + VFH) hasta que la norma de la covarianza era menor que 0.1, y después creaba una ruta para llegar al destino marcado. Por último, el controlador Pure Pursuit guiaba al robot a lo largo de esta ruta. En la siguiente secuencia de imágenes puede visualizarse este comportamiento.

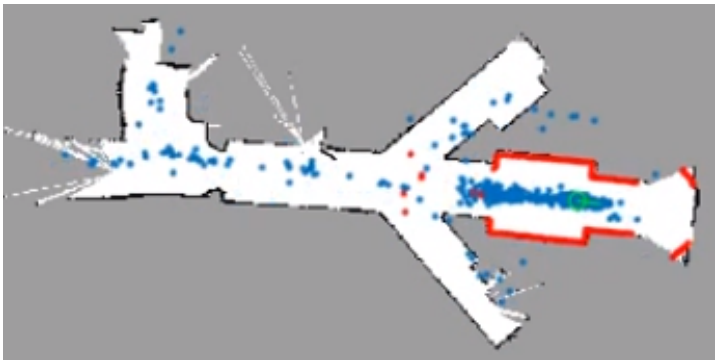

Figura 3.44: Integración: Paso 1.

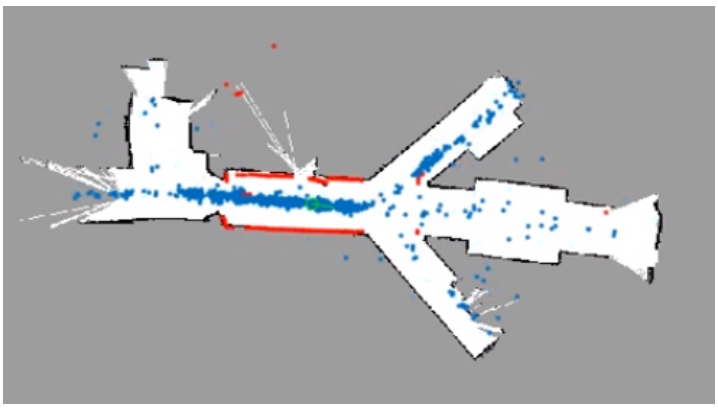

Figura 3.45: Integración: Paso 2.

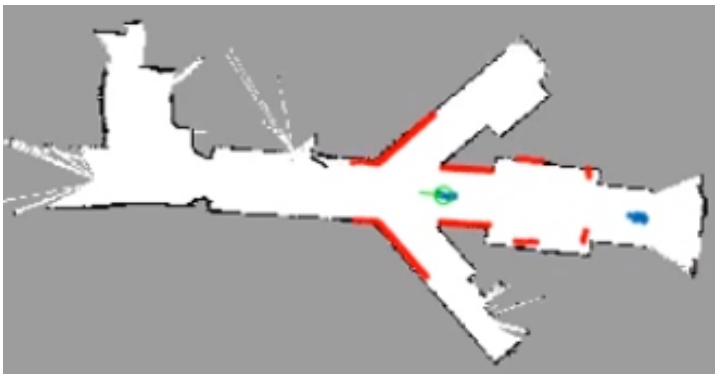

Figura 3.46: Integración: Paso 3.

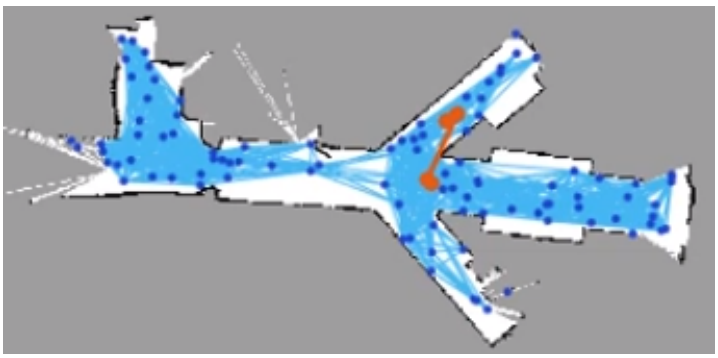

Figura 3.47: Integración: Paso 4.

Tras probar el correcto funcionamiento en simulación, se pasó a implementarlo en las plataformas reales. Como era de esperar, los resultados son similares a los observados en la aplicación de planificación global. Hay una gran diferencia entre los resultados con el Amigobot y con el Seekur. El Amigobot presenta una respuesta bastante satisfactoria, pero mucho menos robusta que en simulación. Por otro lado, con el Seekur no se consiguió encontrar una combinación de parámetros óptima. Este presenta un comportamiento aleatorio, fallando en llegar a su destino en la mayoría de ocasiones. Por tanto, podemos determinar que la aplicación global no es aplicable al Seekur, ya que falla en numerosas ocasiones.

Sin embargo, como el objetivo principal es la aplicación de estos algoritmos en el ámbito docente, y dado que en este se trabaja con robots pequeños como el Amigobot, se considera que se han alcanzado los objetivos propuestos inicialmente. En el capítulo 4.1 se analizan las conclusiones a las que se ha llegado con la realización de este proyecto.

Con el fin de poder mostrar los resultados obtenidos durante el desarrollo de este proyecto, se adjuntan vídeos[5] de todos los algoritmos implementados tanto en simulación como en las dos plataformas reales, en los que se muestran los gráficos de la aplicación a la vez que el comportamiento del robot.

[5]El enlace a los vídeos se encuentra en el anexo Planos, en el apartado C.3.

3.7 Guiado de vehículos en entorno CARLA - ROS.

Una vez alcanzados los objetivos iniciales del proyecto, y vista la escasa robustez del algoritmo de control empleado junto con el PRM, el controlador *Pure Pursuit*, se decidió ampliar el trabajo mediante la creación de un controlador propio más robusto basado en control óptimo, el cual fue depurado y testeado usando el novedoso simulador CARLA explicado en la sección 2.4.

Dado que se trata de un controlador más complejo y preciso, el uso de una plataforma CARLA-ROS permite testear mejor su funcionamiento.
Además, la participación en el CARLA Challenge supuso una motivación extra para la realización de este cometido utilizando este novedoso simulador.

3.7.1 Generación de trayectoria.

Necesitamos un controlador basado en el seguimiento de waypoints pre-computarizados, por lo que surge la necesidad de crear primero una aplicación que, a partir de dichos waypoints, cree una trayectoria. Esta aplicación será codificada en el nodo *spline_node.cpp*. Para ello se implementa un algoritmo de interpolación mediante splines parametrizadas, que representan una curva Q en el espacio bidimensional (x,y) en función de un parámetro u (parámetro de avance a lo largo de la curva): `Q(u)=(X(u), Y(u))` donde X(i) representa las coordenadas x de la curva en función del parámetro u, y Y(u) representa las coordenadas y, cómo se muestra en la figura 3.48.

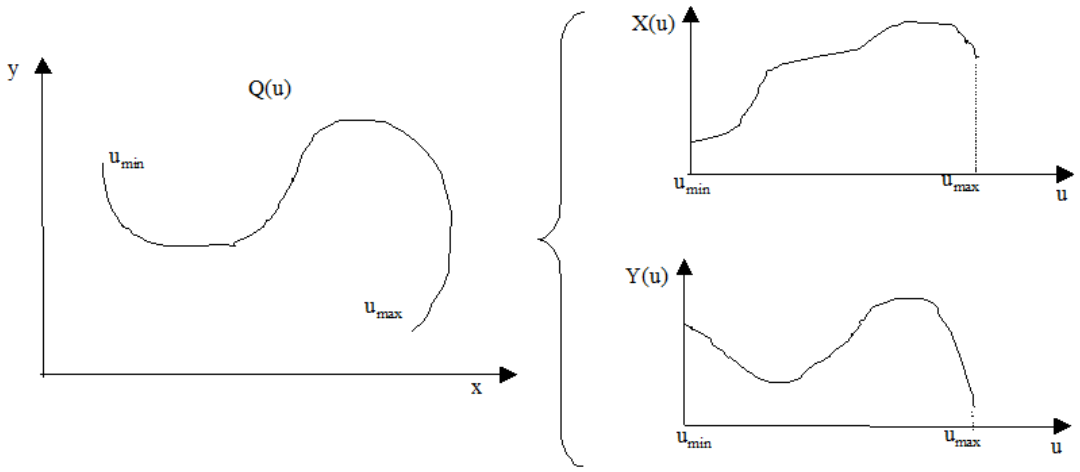

Figura 3.48: Spline parametrizada: Q(u).

Las funciones X(u) e Y(u) más fáciles de implementar son polinomios. Puesto que para la aplicación desarrollada es interesante definir una serie de puntos intermedios por los que se desea que pase la curva, se dividirá ésta en varios tramos, cada uno de ellos enlazando dos puntos consecutivos, y siendo cada uno de ellos un polinomio de tercer grado. Para que el cálculo de cada tramo sea independiente del parámetro u, se normalizará éste de manera que en cada tramo, su valor inicial sea u=0 y su valor final u=1. De este modo, cada tramo i estará definido por dos polinomios de tercer grado.

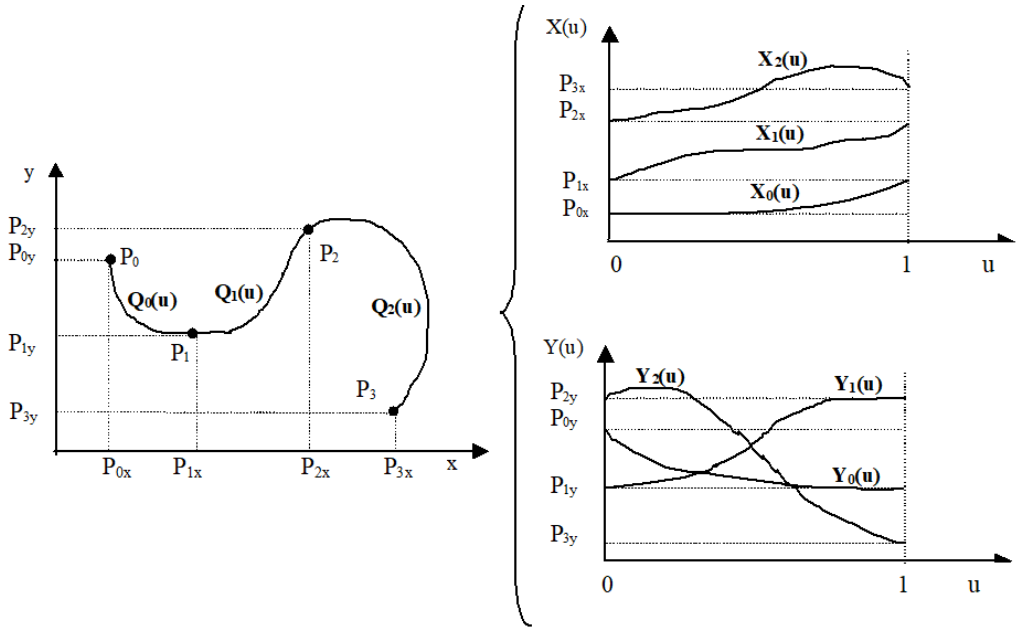

Figura 3.49: Spline parametrizada: Representación de los tramos de Q(u).

El cálculo de los coeficientes se realiza imponiendo una serie de condiciones. Estas condiciones son: continuidad en la primera y segunda derivada en todos los puntos intermedios de la spline, y además asegurar una determinada pendiente de la curva en sus extremos. A continuación se expone el algoritmo necesario para calcular los 4 coeficientes de cada tramo de las dos curvas X(u) e Y(u).

$$Y_i(u) = a_{iy} + b_{iy}u + c_{iy}u^2 + d_{iy}u^3 \tag{3.1}$$

con $0 \leq u \leq 1$

Las condiciones que debe cumplir dicho polinomio son:

1. Pasar por los puntos que enlaza, es decid P_iy y $P_{(i+1)y}$:

$$Y_i(0) = P_iy = a_{iy} \tag{3.2}$$

$$Y_i(1) = P_{(i+1)y} = a_{iy} + b_{iy} + c_{iy} + d_{iy} \tag{3.3}$$

2. Las derivadas primeras en los puntos que enlaza deben tomar un determinado valor (aquél que asegura continuidad en primera y segunda derivada en los puntos intermedios, o bien un valor determinado en los puntos extremos):

$$Y_i^1(0) = D_iy = b_{iy} \tag{3.4}$$

$$Y_i^1(1) = D_{(i+1)y} = b_{iy} + 2c_{iy} + 3d_{iy} \tag{3.5}$$

Las cuatro ecuaciones anteriores dan lugar a los cuatro coeficientes del polinomio:

$$a_{iy} = P_i y \tag{3.6}$$

$$b_{iy} = D_{iy} \tag{3.7}$$

$$c_{iy} = 3(P_{(i+1)y} - P_{iy}) - 2D_{iy} - D_{(i+1)y} \tag{3.8}$$

$$d_{iy} = 2(-P_{(i+1)y} + P_{iy}) + D_{iy} + D_{(i+1)y} \tag{3.9}$$

Únicamente queda determinar los valores de las derivadas primeras en los puntos por los que pasa el polinomio.

1. Para los puntos intermedios, la condición que deben cumplir sus derivadas primeras es que tomen el valor apropiado para asegurar continuidad en la segunda derivada de los polinomios que enlazan. Es decir:

$$Y^2_{i-1}(1) = Y^2_i(0)2c_{(i-1)y} + 6d_{(i-1)y} = 2c_{iy} \tag{3.10}$$

   y sustituyendo los coeficientes de las ecuaciones 3.6 obtenemos:

$$D_{(i-1)y} + 4D_{iy} + D_{(i+1)y} = 3(P_{(i+1)y} - P_{(i-1)y}) \tag{3.11}$$

2. Para los puntos extremos, hay que imponer sus pendientes. Para ello se parte de un valor de orientación inicial de la curva Q(u) y un valor de la orientación final de la curva Q(u). Esas orientaciones inicial y final deben ser traducidas a pendientes (o derivadas) inicial y final de los polinomios $X_0(u), Y_0(u) y X_{(n-1)}(u) y Y_{(n-1)}(u)$, siendo n el número de tramos de la spline.

Así, en el primer punto P_0 sabemos que la pendiente de la curva $Q_0(u)$ debe ser la tangente de la orientación inicial θ_{ini}:

$$tg(\theta_{ini}) = \frac{\mu sen(\theta_{ini})}{\mu cos(\theta_{ini})} \tag{3.12}$$

donde cuanto mayor sea el parámetro μ, la pendiente impuesta afectará a más valores de u (más suave).

Así mismo, en el último punto la pendiente de la curva $Q_{n-1}(u)$ debe ser la tangente de la orientación final:

$$tg(\theta_{fin}) = \frac{\mu sen(\theta_{fin})}{\mu cos(\theta_{fin})} \tag{3.13}$$

Por tanto, al sistema de ecuaciones para determinar los coeficientes del polinomio $Y_0(u)$ de la spline hay que añadir el valor de la primera derivada en el punto P_0 y del mismo modo, al sistema de ecuaciones para determinar los coeficientes del polinomio $Y_{n-1}(u)$ hay que añadir el valor de la primera derivada en el punto P_n.

Al existir (n+1) puntos, hay que determinar sus derivadas mediante un sistema de (n+1) ecuaciones, que puede simplificarse mediante el primer bucle del código mostrado a continuación. El segundo bucle realiza las mismas operaciones en el vector de términos independientes. El último bucle se encarga del cálculo final para resolver las derivadas.
Una vez conocidas las derivadas primeras en todos los puntos, se aplican directamente las ecuaciones 3.6 a cada tramo para obtener sus coeficientes.

```
lam[0]=0;
for (int i=1; i<=(m-2); i++)
{
  lam[i]=1/(4-lam[i-1]);
}

deta[0]=nu*sin(o[0]);
for (int i=1; i<=(m-2); i++)
{
  deta[i]=(3*(y[i+1]-y[i-1])-deta[i-1])*lam[i];
}
deta[m-1]=nu*sin(o[1]);

D[m-1]=deta[m-1];
for(int i=(m-2);i>=0;i--)
{
  D[i]=deta[i]-lam[i]*D[i+1];
}
```

Tras haber explicado el algoritmo de interpolación usado, se describirá la codificación del mismo dentro del nodo *spline_node.cpp*.

Este nodo deberá subscribirse a la trayectoria descrita por los waypoints y a la odometría del móvil. Como primer paso se publicará la trayectoria interpolada, que será del tipo `nav_msgs::Path`.

```
spline\_pub = n.advertise<nav\_msgs::Path> ("trajectory\_spline",1,
   true);
```

Para asegurarnos de que no intenta calcular la trayectoria antes de recibir la odometría (dado que se necesita la orientación inicial para la interpolación) se crea el flag *init_odom* que se pondrá a 1 al final del callback de la odometría. Un callback no es más que una función en la que se entra cuando se recibe un dato del topic al que nos hemos suscrito. Esta función es indicada al crear el suscriptor:

```
ros::Subscriber spline\_sub = n.subscribe("trajectory\_vis", 1000,
   trajectoryCallback);

ros::Subscriber odom\_sub = n.subscribe('odom', 1000,
   odometryCallback);
```

siendo *trajectory_vis* el topic que publica los waypoints y *odom* el topic que contiene los datos de odometría.

En *trajectoryCallback* unicamente se recoge el dato de odometría (x,y,yaw) en *actualPose* y se pone a 1 el flag *init_odom*.

En *trajectoryCallback* se llevan a cabo los siguientes pasos:

1. Se comprueba que la trayectoria ha cambiado. Si no es así, se sale de la función para no estar calculándola continuamente, dado que los waypoints se publican constantemente.

2. Como primer punto de la trayectoria cogemos la posición actual del coche (*points_spline[0]*).

3. Dentro de un bucle, recorremos los m waypoints. Si respecto al waypoint anterior escogido se supera una distancia *N_distancia* se añade el punto al vector *points_spline*.

4. En el vector o, se recogen la orientación inicial del coche de la odometría, y la final como la tangente de los dos puntos finales de la curva.

5. Una vez recogidos los puntos se llama a la función *n_spline* pasándole como parámetros los vectores x, y y o, correspondientes a la posición y orientación de estos respectivamente, así como un valor de $\mu = 0{,}0001$ y el número de puntos *n_puntos*.
   En esta función se realiza el cálculo de los coeficientes de la spline como se explicó anteriormente.

6. Dentro de un bucle se recorren todos los tramos de la trayectoria y a su vez, dentro de cada tramo, los 10 tramos de la curva desde u=0 hasta u=1. En este se rellena el mensaje *spline_path*, calculando la posición de cada punto de la curva con la ecuación descrita en 3.1 para x e y. Una vez rellenado el mensaje, se publica con: `spline_pub.publish(spline_path);`

3.7.2 Control para el seguimiento de trayectoria.

Habiendo calculado la trayectoria a seguir por nuestro vehículo ya interpolada, se procederá a diseñar el controlador dentro del mismo nodo *spline_node.cpp*. Para que el controlador no empiece a funcionar hasta que no se publique una trayectoria, se creará otro flag *init*, que se pondrá a 1 una vez esta sea publicada. Dado que el controlador tiene que funcionar periódicamente, este se implementa dentro del callback del timer, que retornará inmediatamente sin ejecutar ninguna instrucción hasta que este flag esté activo.

El controlador estará basado en un control óptimo lineal cuadrático (LQR) [62] con predicción que determine la velocidad angular del vehículo. La velocidad lineal se calculará como una simple relación de proporcionalidad entre la velocidad máxima y la curvatura de la siguiente forma:

```
v=(v_max*rc)/rc_max;
```

Veamos ahora brevemente en qué consiste este tipo de controladores.
Estos permiten obtener sistemas eficientes, optimizando el proceso de prueba y error de los controladores clásicos como el PID, el cual se diseña mediante un proceso empírico. Se basan en encontrar un principio óptimo que satisfaga las restricciones físicas y minimice o maximice ciertos criterios de desempeño del controlador. En nuestro caso buscamos minimizar dos errores en la ubicación del automóvil: el error lateral y el error de orientación. El error lateral se trata de la distancia desde la ubicación actual hasta el punto de la trayectoria más cercano.

La figura 3.50 muestra un ejemplo de los errores lateral y de orientación para una configuración concreta del robot.

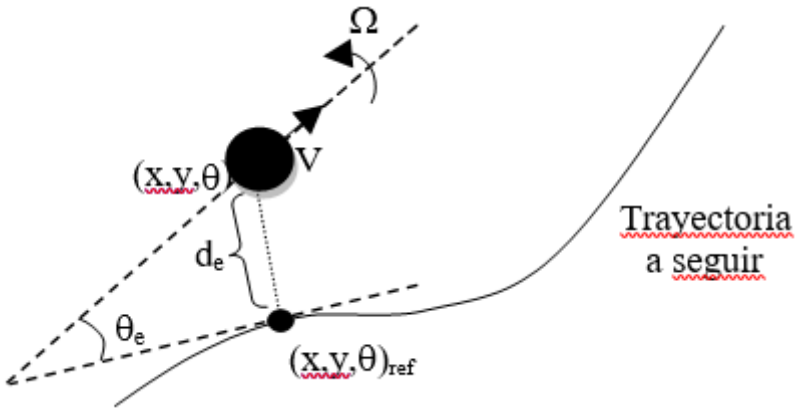

Figura 3.50: Esquema de los errores lateral y de orientación.

Como se observa, el error lateral d_e se define como la distancia desde la ubicación real hasta el punto más cercano de la trayectoria, lo cual implica que es perpendicular a la tangente a la trayectoria en dicho punto, es decir, a θ_d. Esto, y el hecho de que pueda tomar signo positivo o negativo según se encuentre por encima o por debajo de la trayectoria, hace que el error lateral, junto con el de orientación, sea suficiente para caracterizar de forma exacta el error en la ubicación del coche.

Tras representar el sistema en variables de estado, es necesario obtener el valor de la matriz de ganancias K para obtener la respuesta deseada. El control LQR se basa en la elección de una matriz K que minimice una función de coste cuadrática que depende de las matrices de ponderación Q y R. Estas determinan la importancia relativa entre el estado y la entrada de control. Como nuestro vector de estados se compone de los dos errores definidos, Q será una matriz 2x2 de manera que q_{11} determina la importancia relativa del error lateral y q_{22} del error de orientación. R será un escalar que determina la importancia relativa de la entrada de control, la velocidad angular.

El hecho de incluir en este controlador una ventana de predicción se debe a la alta velocidad que puede llegar a alcanzar un automóvil y la necesidad de anticiparse para poder realizar un control óptimo y eficiente.

Conocidos los principios del controlador a implementar se procede a codificar el mismo.

Periódicamente, cada vez que interrumpa el timer, se ejecutan los siguientes pasos:

1. Propagación a *futurePose* de la posición actual. Dentro de un bucle se predice la posición del vehículo en *N_RET_VEL* iteraciones:

```
futurePose[0] = actualPose[0];
futurePose[1] = actualPose[1];
futurePose[2] = actualPose[2];

for (int i=0; i<N_RET_VEL;i++)
{
  futurePose[0] = futurePose[0] + evol_vel[0][i]*cos(
     futurePose[2])*sample_time;
  futurePose[1] = futurePose[1] + evol_vel[0][i]*sin(
     futurePose[2])*sample_time;
  futurePose[2] = futurePose[2] + evol_vel[1][i]*
     sample_time;
  // ROS_INFO("\nVelocidades: %lf %lf --> poscion predicha:
      %lf %lf %lf", evol_vel[0][i],evol_vel[1][i],
     futurePose[0], futurePose[1], futurePose[2]);
}
```

2. Cálculo de la posición de consigna. Esta posición coincide con el punto más cercano de la posición predicha del automóvil a la trayectoria y su orientación. Este paso lo realiza la función *calcula_consigna(futurePose, coefs, n_tramo)*.

3. Publicación de los mensajes *pose_consigna* y *pose_future*.

4. Cálculo de errores lateral y de orientación utilizando la consigna y la posición predicha:

```
de = (futurePose[1]-pose_d[1])*cos(pose_d[2])-(futurePose[0]-
   pose_d[0])*sin(pose_d[2]);
oe = Limitang(futurePose[2]-pose_d[2]);
```

5. Cálculo de la velocidad lineal proporcional a la velocidad máxima (10m/s) y la curvatura.

6. Aplicación de la ley de control óptimo para obtener la velocidad angular:

```
      A[1]= v*sample_time;
      B[0]= v*(sample_time*sample_time)/2;
      Dlqr(Kr,A,B,Q,R);
      Consigna_W(de,oe,Kr,&w);
```

   Con el modelo cinemático del móvil creamos las matrices A y B. Empíricamente se asigna un valor a las matrices de ponderación de:

```
  Q[4]={1000,0,0,0.0001}, R=50000;
```

   La función `Dlqr` implementa las operaciones necesarias para calcular el valor de la ganancia Kr.

7. Publicación de los mensajes *steer_msg* con el valor de la velocidad angular y *speed_msg* con el valor de la velocidad lineal deseada.

8. Actualización del buffer de velocidades con el comando de velocidad publicado en *speed_msg*.

3.7.3 Percepción de la escena y comportamiento reactivo.

Una vez creado el nodo que genera la trayectoria y el control del vehículo, se procede a la codificación del nodo *object_detection.cpp*, el cual recibirá y procesará datos provenientes de un lidar y una cámara con el fin de detectar semáforos y objetos.

Para la detección de los semáforos se hace uso de una red neuronal ya entrenada, YOLO [67] [68]. Empleando la función *objetos_yolo*, ya implementada, para detectar semáforos verdes y rojos.

En *xyz_filter* se lleva a cabo el procesado de los datos del láser para detectar objetos. La función del primer bucle es quedarse con una sección de estos datos en la que se buscarán los objetos. Empíricamente se hallaron las medidas respecto al centro del coche de la sección de interés:

- `Eje z o altura`: -1m <z <1.5m, altura justa para no detectar el suelo.

- `Eje y o ancho`: -1.1m <y <1.5m, ancho justo para no detectar a los coches del carril adyacente o bancos y otros objetos de la acera. Se detectan sólo los objetos del propio carril así como aquellos objetos de la acera que sobresalgan hacia la carretera y peatones o ciclistas que estén a punto de cruzar.

- `Eje x o largo`: 2m <x <17m. Se queda con los datos a partir de 2m para no detectar el propio coche. Se observó que una distancia de 17m es adecuada para reaccionar con tiempo a los obstáculos.

Así mismo, dentro de este `if`, se guarda la distancia al objeto más cercano.

En vista de que ya sabemos si existe algún semáforo y la distancia al objeto más cercano, se rellena el topic *obstacle*, que por defecto valdrá 0. Este será un código que indique el grado de aceleración o freno que debe aplicarse para no chocar con el objeto. Dado que para ello debemos tener en cuenta si nos encontramos en una recta o en una curva, nos suscribimos al topic *steer_cmd* publicado por *spline_node*. El algoritmo seguido es el siguiente:

1. Si nos encontramos en una recta:

   (a) Si la distancia al objeto más cercano está entre 10m y 17m se envía el código 2, que significa que debe acelerar poco de manera que se frene por rozamiento. Así mismo se va aumentando el valor de un contador, de manera que si este llega a 20 se cambie el código a un 3 para que se acelere más y el coche empiece a moverse. Esto impide que se quede atascado ante un objeto estático.

   (b) Si la distancia es menor de 10m se envía el código 1, que significa que se debe parar de manera brusca ya que en recta la velocidad será alta. Igualmente se vuelve a aumentar el valor de un contador. Cuando el valor de este llegue a 600 se envía el código 3 para que empiece a moverse el vehículo.

2. Si nos encontramos en una curva:

   (a) Solo paramos si la distancia es menor de 3m, haciendo una parada brusca (*object*=1). Se utiliza un contador de 600.

3. Si la función *objetos_yolo* ha detectado un semáforo rojo se envía *object* = 1.

3.7.4 Agente para la participación en el CARLA Challenge.

El siguiente paso tras haber comprobado el funcionamiento de este nodo junto con *spline_node*, es la creación del agente que será presentado al concurso.
Como punto de partida se estudiaron las bases del CARLA Challenge.

En este los agentes conducirán a lo largo de una serie de rutas predefinidas, las cuales serán las mismas para todos los equipos. Para cada ruta, los agentes se inicializan en un punto inicial y deberán conducir hasta un destino concreto como se muestra en la figura 3.51, dado por una descripción a alto nivel de la ruta en forma de instrucciones ([sigue recto, gira a la izquierda,

fin]), etc. Estas instrucciones representan las decisiones que el agente necesita para cruzar intersecciones o rotondas. Además en nuestro caso, al ser participantes del Track3, se reciben una serie de waypoints que codifican posiciones GPS representando la ruta.

Figura 3.51: Ilustración de una ruta, marcando en verde la trayectoria, punto inicial en azul y punto destino en rojo.

El challenge ofrece cuatro tracks paralelos, que se diferencian en los datos de entrada de los que se provee a los agentes. Nuestro Track está pensado para equipos que quieran automatizar la conducción basándose en el uso de mapas y waypoints ya pre-computarizados. Por ello, tenemos acceso a los siguientes sensores o pseudo-sensores: HD Map + Waypoints + LIDAR + GPS + Cámaras RGB.
Además, tenemos a nuestra disposición un plan topológico de alto nivel *_global_plan* codificado como una lista de tuplas. Cada una de estas tuplas contiene una localización GPS y una acción: *RoadOption.LEFT, RoadOption.RIGHT, RoadOption.STRAIGHT, RoadOption.LANEFOLLOW, RoadOption.CHANGELANELEFT, RoadOption.CHANGELANERIGHT.*

Cada ruta será repetida un numero de veces bajo diferentes condiciones meteorológicas y de tráfico. Las rutas usarán distintos mapas, incluyendo autopistas, escenarios urbanos y distritos residenciales. Por tanto, los agentes deben saber afrontar situaciones de tráfico relacionadas con estos entornos, como: unificación de carriles, cambios de carril, negociaciones en intersecciones y rotondas, cumplimiento de señales y semáforos, respetar a peatones, ciclistas y otros elementos, etc.

El rendimiento de un agente dependerá del número de rutas completadas satisfactoriamente. Una ruta es considerada como tal si no se comete ninguna infracción crítica. Si es así, el episodio termina automáticamente y el agente recibe la puntuación proporcional al porcentaje de la ruta completada hasta dicha infracción. Igualmente, si se acaba el tiempo establecido para completar dicha ruta la puntuación será el porcentaje completado de dicha ruta. Por tanto, la puntuación final dependerá de dos factores: los puntos obtenidos por la ruta recorrida y los puntos perdidos por infracciones.

Una vez conocidas las bases del concurso, se pasa a instalar y probar las herramientas, para posteriormente implementar nuestros algoritmos haciendo uso de ellas. Primero hacemos un esquema de cómo será nuestro sistema, la comunicación entre los diferentes nodos y el simulador CARLA.

En la figura 3.52 se muestran los tres nodos de ROS, el agente *RobesafeAgent.py* y el simulador. Más adelante se explicará más en detalle la funcionalidad de cada topic.

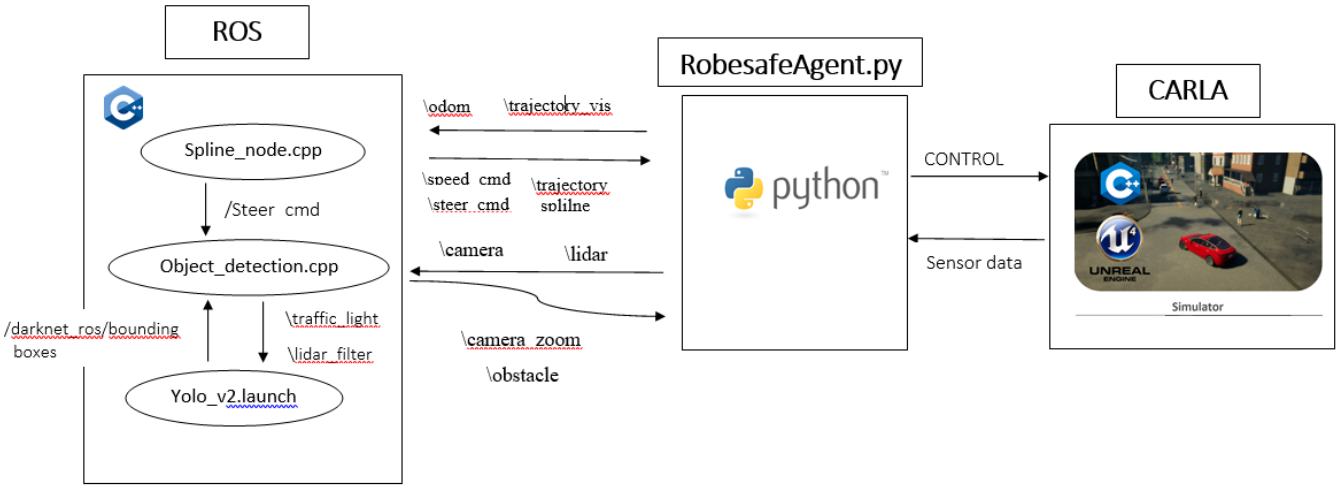

Figura 3.52: Esquema de la arquitectura de trabajo CARLA-ROS.

A continuación se detallan los pasos seguidos para la puesta en marcha de nuestro sistema.

1. Descarga e instalación del simulador CARLA así como del repositorio *ScenarioRunner* desde *GitHub*.

2. Para hacer una primera prueba, se procede a ejecutar uno de los agentes de ejemplo contenidos en el repositorio *ScenarioRunner*. La idea para la evaluación es puntuar al agente en situaciones de tráfico diferentes descritas en ficheros *.json*. Un escenario está definido por una trayectoria y ciertos eventos que tendrán lugar durante el recorrido. El escenario también controla el criterio de finalización y puntuación.
   Primero lanzamos el servidor de CARLA con los parámetros correctos:

```
./CarlaUE4.sh -benchmark -fps=20 -quality-level=Epic
```

   Para esta prueba se elige usar el agente de ejemplo *HumanAgent.py* el cual es controlado manualmente con teclado. Para ejecutar el agente se usa la siguiente sintaxis:

```
Python ${ROOT_SCENARIO_RUNNER}/srunner/challenge/
   challenge_evaluator_routes.py \
--scenarios=${ROOT\_SCENARIO\_RUNNER}/srunner/challenge/
   all_towns_traffic_scenarios1_3_4.json \
--routes=${ROOT_SCENARIO_RUNNER}/srunner/challenge/
   routes_training.xml \
```

```
    --repetitions=3\--debug=0\--agent=\${TEAM\_AGENT} \
    --config=${TEAM_CONFIG}
```

Para lo cual es necesario crear antes la variable de entorno `TEAM_AGENT`:

```
    TEAM_AGENT=${ROOT_SCENARIO_RUNNER}/srunner/challenge/autoagents
       /HumanAgent.py
```

3. El siguiente paso será la creación del agente. Para ello hay que definir una clase agente que sea heredada de la clase base *AutonomousAgent*. En nuestra clase agente, hay tres funciones principales que han de ser definidas para poder ejecutarlo:

   - `Metodo Setup`: este método es el lugar en el que se realizan todas las inicializaciones de las variables de control y definiciones necesarias. Es llamado automáticamente cuando se crea una instancia del agente. Es aquí donde se declara el Track en el que se participará:

     ```
           self.track = Track.ALL_SENSORS_HDMAP_WAYPOINTS
     ```

     Además en este método se llama a los nodos *spline_node*, *object_detection* y *yolo_v2.launch* mediante el uso de `os.system`.

   - `Método Sensors`: en este método se definen los sensores requeridos por el agente. La sintaxis es la siguiente:

     ```
     def sensors(self):
       sensors = [{'type': 'sensor.camera.rgb', 'x':0.7, 'y':0.0, 'z
          ':1.60, 'roll':0.0, 'pitch':0.0, 'yaw':0.0, 'width':600, '
          height':400, 'fov':100, 'id': 'Center'},
       {'type': 'sensor.camera.rgb', 'x':0.7, 'y':-0.4, 'z': 1.60, '
          roll': 0.0, 'pitch': 0.0, 'yaw': -45.0, 'width': 600, '
          height': 400, 'fov': 100, 'id': 'Left'},
       {'type': 'sensor.camera.rgb', 'x': 0.7, 'y':0.4, 'z':1.60, '
          roll':0.0, 'pitch':0.0, 'yaw':45.0, 'width':600, 'height'
          :400, 'fov': 100, 'id': 'Right'},
       {'type': 'sensor.camera.rgb', 'x': -1.8, 'y': 0, 'z': 1.60, '
          roll': 0.0, 'pitch': 0.0, 'yaw': 180.0, 'width': 600, '
          height': 400, 'fov': 130, 'id': 'Rear'},
       {'type': 'sensor.lidar.ray_cast', 'x': 0.7, 'y': 0.0, 'z':
          1.60, 'roll': 0.0, 'pitch': 0.0, 'yaw': 0.0, 'id': 'LIDAR'
          },
       {'type': 'sensor.other.gnss', 'x': 0.7, 'y': -0.4, 'z': 1.60,
           'id': 'GPS'},
     ```

```
    {'type': 'sensor.can_bus', 'reading_frequency': 25, 'id': '
        can_bus'},
    {'type': 'sensor.hd_map', 'reading_frequency': 1, 'id': '
        hdmap'},
    ]

return sensors
```

Cada sensor es un diccionario en el que se ha de especificar el tipo de sensor, el id o etiqueta que se usará para acceder al sensor y otros parámetros dependientes del sensor como el *fov* en las cámaras. Se pueden establecer parámetros intrínsecos y extrínsecos (localización y orientación), en coordenadas relativas al vehículo, siendo z la altura, x adelante-atrás e y izquierda-derecha. Los sensores utilizados son:

- `Sensor.camera`: cámara RGB
- `Sensor.lidar`: LIDAR Velodyne 32
- `Sensor.can_bus`: CANBus con información de odometría como velocidad y orientación del vehículo.
- `Sensor.other.gnss`: GPS.
- `Sensor.hd_map`: mapa HD cargado y posición actual del vehículo para habilitar la localización en el mapa.

- `Método run-step`: este método se llama automáticamente en cada paso de simulación y recibe datos de entrada como parámetro. Estos datos de entrada son un diccionario con todos los sensores especificados en la función *sensors*. Esta función es la encargada de retornar el control a ser aplicado al vehículo. Los comandos de control que se deben enviar al coche son aceleración y dirección o giro de volante. Se compara el comando de velocidad recibido de *spline_node.cpp* `cmd_speed` con la velocidad actual leída del pseudo-sensor *hd_map*, de manera que si la diferencia es:
  - Mayor de 7m/s: se aplica una aceleración de $1m/s^2$.
  - Menor de 7m/s y mayor de 3m/s: se aplica una aceleración de $0{,}8m/s^2$.
  - Menor de 3m/s y mayor de 0.3m/s: se aplica una aceleración de $1m/s^2$.
  - Menor de 0.3m/s: no se aplica aceleración.

  Así mismo, es en este método en el que se aplica el tratamiento de obstáculos para dotar al vehículo de un comportamiento reactivo:
  - Si *obstacle* = 1 debemos hacer una parada brusca, por lo que se suelta el acelerador y se activan el freno y el freno de mano.
  - Si *obstacle* = 2 se aplica una aceleración = $0{,}5m/s^2$ de manera que se frene poco a poco por rozamiento.
  - Si *obstacle* = 3 se empieza a acelerar poco a poco aplicando una aceleración = $0{,}5m/s^2$.

Además al principio de la ejecución se establece la ruta completa que ha de seguir el agente en la variable `self.global_plan`. Esta es representada como una lista de tuplas, conteniendo

los waypoints de la ruta, expresados como latitud y longitud a lo largo de la opción de ruta recomendada. Para una intersección, las opciones pueden ser seguir recto, girar a izquierda o girar a derecha. En carreteras con múltiples carriles, las opciones también pueden ser cambiar al carril de la derecha o al de la izquierda.

4. Una vez creados los métodos imprescindibles del Agente, se procede a entrenar y testear este, para lo que se dispone de una serie de rutas predefinidas (*routes_training.xml* para entrenar, *routes_devtest.xml* para verificar y *routes_test.xml* para depurar en la nube).

   A parte de estos métodos principales, se crea la clase `ControlInterface` para la visualización con *pygame*, inicialización del nodo de ROS, y tratamiento de todos los suscriptores y publicadores que harán de comunicación con los otros dos nodos a través de topics. Estos se listan a continuación:

   - Publicación de *trajectory_vis* que contiene la lista de waypoints incluidos en *self.global_plan*. Este es el topic al que nos suscribimos desde el nodo *spline_node* para interpolar la trayectoria y calcular los comandos de control. La sintaxis para crear un publicador es la siguiente:

   ```
   self.spline_pub = rospy.Publisher('trajectory_vis', Path,
      queue_size=1000)
   ```

   - Publicación de la odometría en el topic *odom*. Al que nos suscribimos desde *spline_node*.
   - Publicación de los datos del láser en *lidar*. A este topic nos suscribimos en *object_detection* para el procesado de los datos del láser y posterior identificación de objetos.
   - Publicación de la imagen de la cámara en *camera*, topic al que nos suscribiremos desde *object_detection* para la detección de los semáforos.
   - Suscripción al comando de velocidad angular *steer_cmd* publicado desde *spline_node*. La sintaxis para crear un suscriptor es la siguiente:

   ```
   self.steer_sub = rospy.Subscriber('steer_cmd',Float64,
      read_steer)
   ```

     Donde *read_steer* es el callback de dicho suscriptor. Cada vez que llegue un mensaje de este topic se ejecutará dicha función. Lo único que se hará en estos callbacks es leer los datos del topic e introducirlos en una variable global de manera que pueda ser vista desde cualquier clase o función.

   - Suscripción al comando de velocidad lineal *speed_cmd* publicado desde *spline_node*.
   - Suscripción al topic *obstacle* publicado desde el nodo *object_detection* con la información de los objetos de manera codificada como se explicó anteriormente.

   Para inicializar el nodo de ROS y establecer la comunicación con el simulador se utiliza la siguiente sintaxis:

```
    rospy.init_node("agente_py", anonymous=True)
    host = rospy.get_param("/carla/host", "192.168.73.176")
    port = rospy.get_param("/carla/port", 2000)
    rospy.loginfo("Trying to connect to {host}:{port}".format(
    host=host, port=port))
```

Dentro del método *run* de esta clase se recogen los datos de los sensores y se procesa la información de las cámaras para la visualización de la trayectoria durante la ejecución.

Aquí se llama a la función `trajectoryCallback`, donde se realizan todas las publicaciones mencionadas previamente. Primero se procesan los waypoints recogidos de *global_plan* y se rellena el mensaje del topic *trajectory_vis* con estos puntos. Del sensor *hd_map* se recogen los datos de la posición y orientación actual del coche. Tras convertir la orientación a cuaternio se rellena el mensaje *odometry*. Habiendo publicado estos dos primeros mensajes, se publican los datos del sensor del lidar y la cámara.

5. Una vez comprobado el funcionamiento del agente en el equipo local, se procede a probar el agente dentro del docker localmente. Cada equipo debe enviar una imagen docker que contenga el agente usando los modelos de *Dockerfile* proporcionados. Los contenedores de docker irán a los repositorios internos del Carla AD Challenge usando scripts EvalAI.

6. CARLA ofrece funcionalidades de registro y reproducción. Esto significa que es posible grabar registros del estado de simulación y usarlos para propósitos de visualización y depuración. Con esto se puede recoger información del episodio, como el ID del vehículo.

```
$ python show_recorder_file_info.py -f log_debug_track0_route_0001.
   log
```

Con esta sentencia, la salida del script sería similar a la figura 3.53.

```
Create 948: vehicle.lincoln.mkz2017 (1) at (3506.6, 11533.5, 50.0015)
 number_of_wheels = 4
 sticky_control = true
 object_type =
 color = 0,0,0
 role_name = hero
```

Figura 3.53: Análisis de los CARLA *logs*.

En este caso el ID es 948. Tras obtener este ID se puede reproducir el registro siguiendo a dicho vehículo:

```
 $ python start_replaying.py -f log_debug_track0_route_0001.log -c
    948
```

Esto reproducirá una secuencia entera en el servidor de CARLA desde la perspectiva del vehículo seleccionado.

7. Habiendo depurado y testeado el agente tanto en local como en la nube, se envía la imagen docker a la fase de test del concurso.

Los resultados obtenidos fueron altamente satisfactorios. El agente resultó ser bastante robusto y se consiguió alcanzar la cuarta posición en el ranking del Track 3 con una puntuación de 52.63/100, habiendo obtenido el ganador una puntuación de 66.83/10.

Track 3

Rank	Team	Route pts	Infraction pts	Total avg
1	LRM-B	79.97	13.7	66.83
2	B4	77.48	11.87	66.05
3	Koalrac	81.05	20.9	60.47
4	robesafe_uah_spain	60.48	9.9	52.63
5	Marln	48.93	13.67	35.87
6	NCTU AIMMlab	45.38	14.6	31.25
7	MSC	4.4	2.8	2.04
8	Tarzan	4.09	4.4	1.92

Figura 3.54: Resultados del Track 3 del CARLA Challenge.

Esto es un gran logro ya que, de 211 participantes organizados en 69 equipos, únicamente 11 llegaron a mandar un agente funcionando y, de esos 11 equipos, alcanzamos la cuarta posición. Las pérdidas de puntos se debieron principalmente a un problema con la percepción de los semáforos. Sin embargo, el controlador desarrollado durante este proyecto resultó ser muy robusto, siguiendo siempre la ruta dada en *_global_plan*, sin salirse del carril y evitando obstáculos correctamente.

En las figuras 3.56 y 3.57 puede verse la vista en RVIZ de la ruta seguida por el vehículo en dos rutas diferentes, el mapa urbano del simulador CARLA, y la visión de la cámara ampliada con la red YOLO funcionando. La ruta a seguir se representa en verde, la odometría en rojo y la posición actual del vehículo en amarillo. En la figura 3.56 se ve como la red YOLO detecta semáforos y bancos, y en la 3.57 coches y, camiones y motocicletas.

En los siguientes enlaces se pueden encontrar dos vídeos en los que se observa el comportamiento del agente Robesafe en el simulador CARLA:

https://youtu.be/22icW48Z8Vs

https://youtu.be/gncjfWjIQ0s

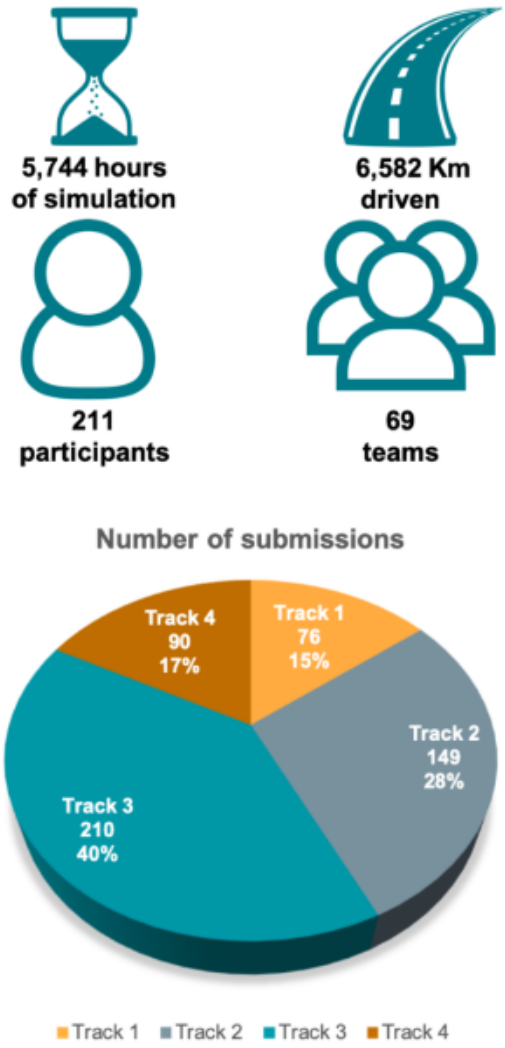

Figura 3.55: Estadísticas CARLA Challenge.

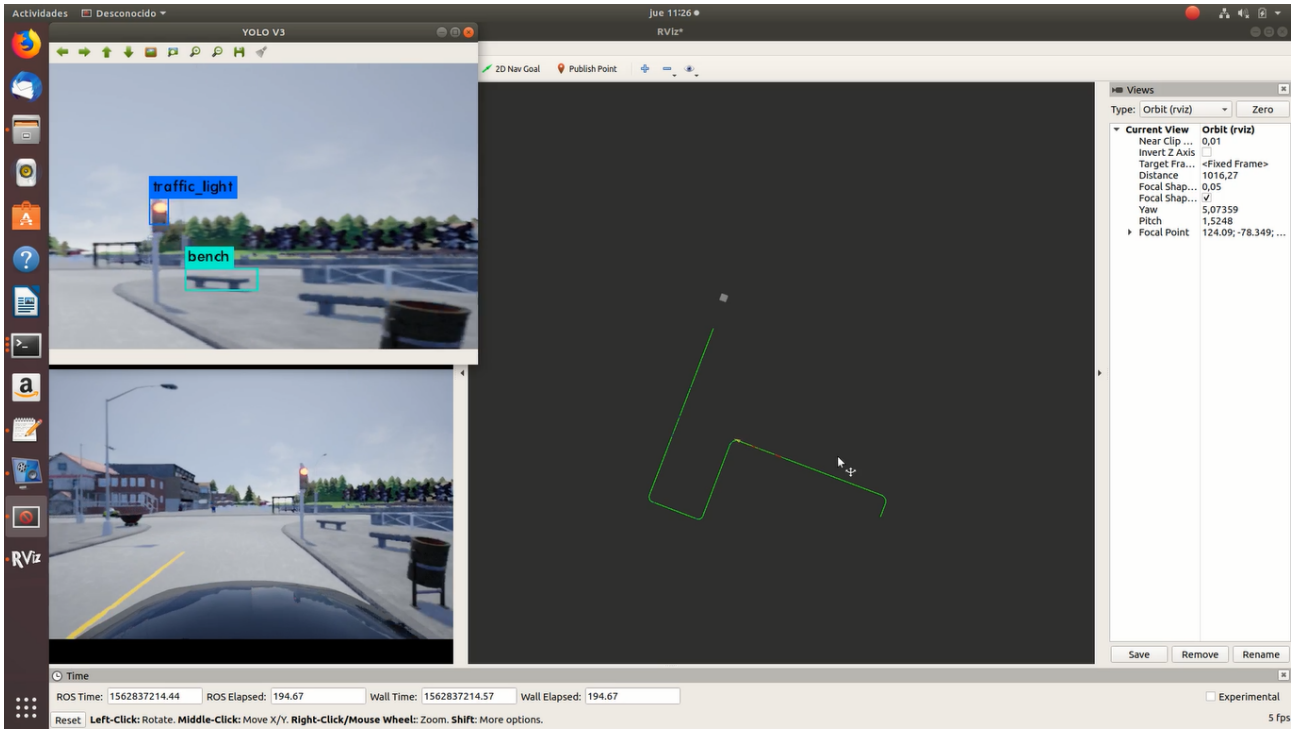

Figura 3.56: Agente RobeSafe.

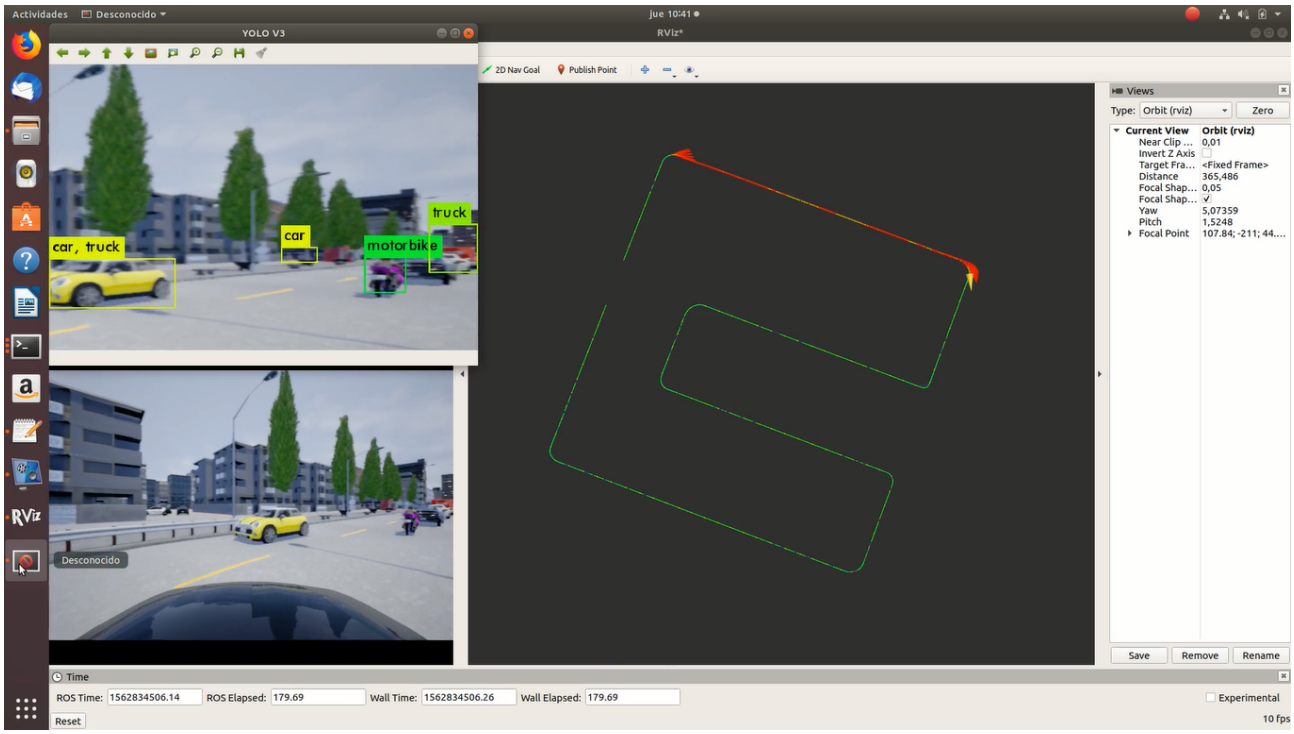

Figura 3.57: Agente Robesafe (2).

Capítulo 4

Conclusiones y Trabajos Futuros

La conquista propia es la más grande de las victorias.

Platón

Para finalizar el proyecto, en este apartado se resumen las conclusiones obtenidas y se proponen futuras líneas de investigación que se deriven del trabajo.

4.1 Conclusiones

Para comenzar con esta sección se analizará si se han conseguido los objetivos marcados en la sección 1.3 dentro del primer capítulo. Para ello, se dividieron los objetivos iniciales en distintos apartados: análisis del entorno de desarrollo, estudio y configuración de la arquitectura del sistema tanto en simulación como con las plataformas robóticas, desarrollo e implementación individual de los distintos algoritmos de navegación en simulación y en los robots, creación y pruebas de una aplicación conjunta de navegación en las dos plataformas robóticas.

Como se ha visto en las secciones 2.2 y 2.3, se hizo un estudio del entorno de desarrollo robótico ROS y de la Robotic System Toolbox de Matlab del cual se han obtenido conocimientos necesarios para la realización de este proyecto. Una vez conocido el entorno de desarrollo, el siguiente paso fue realizar la configuración de la arquitectura del sistema. Para ello se crearon pequeñas aplicaciones iniciales con el fin de comprobar la comunicación eficiente entre las distintas máquinas. Por lo tanto, los primeros objetivos de conocimiento del sistema se han logrado.

Tras esto se fueron desarrollando por orden los diferentes algoritmos de navegación: mapeado, localización, evitación de obstáculos, seguimiento de pasillos y planificación global. Tras comprobar el correcto funcionamiento en simulación y conocer bien el algoritmo en cuestión, se llevaron a cabo los pequeños cambios y ajuste de parámetros necesarios para aplicarlos en las plataformas reales. En general, todos los algoritmos presentaron un funcionamiento más satisfactorio en simulación. Si bien es cierto que en el robot Amigobot todos ellos, en mayor o menor medida, fueron estables, la plataforma Seekur resultó ser mucho más inestable llegando a no conseguir

un correcto funcionamiento en la aplicación global. Esto se atribuyó la escasa fiabilidad del controlador *Pure Pursuit* así como del algoritmo PRM, que en muchas ocasiones no conseguía calcular una ruta conveniente para el robot. Así mismo, la localización con el AMCL resultó ser bastante ineficiente para el robot Seekur.

Dado que se buscaba como objetivo que estos algoritmos fueran aplicables en el ámbito docente, para asignaturas como *Robótica Móvil*, se considera cumplido dicho objetivo ya que las plataformas robóticas usadas en estas son Amigobots. Gracias al exhaustivo estudio de las clases y métodos de la Robotic System Toolbox utilizados para implementar los distintos algoritmos, se han podido comparar y comprobar la fiabilidad de diferentes aplicaciones. De este manera, se incluirán las más eficientes dentro de las asignaturas con el fin de que los alumnos puedan realizarlas sin problemas y con rapidez suficiente para poder incluirlas dentro de una asignatura cuatrimestral.

Una vez conseguidos los objetivos iniciales del proyecto, se decidió ampliar el trabajo mediante la creación de una aplicación de control de trayectoria basado en el seguimiento de waypoints pre-establecidos. Esta aplicación fue depurada y probada en el novedoso simulador CARLA, principalmente pensado para vehículos autónomos.

Para ello se creó primeramente una aplicación que, interpolando los waypoints, generase la trayectoria a seguir por el vehículo. Tras esto se diseñó un controlador óptimo predictivo que proporciona la velocidad angular óptima para el vehículo, mientras que la velocidad lineal se calcula directamente como una relación de proporcionalidad entre la velocidad máxima y la curvatura de la trayectoria.

Dado que el comportamiento de la aplicación era muy satisfactorio se decidió participar con esta en el CARLA Challenge dentro del grupo de investigación Robesafe del Departamento de Electrónica. Como este consistía en una serie de escenarios urbanos que presentaban diferentes retos como el cumplimiento de señales y semáforos y evitación de peatones o ciclistas se hizo necesaria la percepción de la escena. Con la colaboración de otros integrantes del grupo se creó una aplicación conjunta que incorporaba percepción a mediante la red neuronal pre-entrenada YOLO. Aunque para la detección de peatones y ciclistas fue suficiente la utilización de los datos recibidos del láser, con la percepción a través de los datos recibidos de las cámaras y la red neuronal YOLO, se consiguió detectar señales y semáforos. De esta manera se amplió la aplicación inicial para dotar al vehículo de un comportamiento reactivo ante cualquier tipo de imprevisto en la escena.

Este último objetivo supuso un verdadero reto ya que surgió la necesidad de trabajar con un nuevo simulador que hubo que estudiar y, a parte de la tarea de crear una aplicación de navegación eficiente para vehículos autónomos, fue necesario implementar esta como un agente de Python dentro de la compleja estructura que presentaba el concurso. Además, hubo que aprender a trabajar con docker ya que era obligatorio presentar dicho agente dentro de una imagen docker.

Sin embargo, se obtuvieron frutos del esfuerzo, ya que se consiguió finalmente participar en el concurso con un agente bastante robusto, alcanzando la cuarta posición en el ranking.

En conclusión, no solo se alcanzaron los objetivos iniciales, si no que se consiguió ampliar el trabajo provechosamente.

4.2 Líneas futuras

Tanto los objetivos principales del proyecto como la ampliación del trabajo relacionada con el CARLA Challenge, podrían dar lugar a nuevas líneas de investigación.

Primeramente, siguiendo el objetivo inicial del proyecto, sería interesante crear prácticas de laboratorio en las que los alumnos tengan que llevar a cabo estas aplicaciones de manera guiada. Así, podrían aprender de una forma muy visual y práctica los distintos métodos y algoritmos aprendidos en clase de manera teórica.

Así mismo, sería conveniente sustituir las aplicaciones de localización (AMCL) y planificación global (PRM) por otras más novedosas y eficientes, ya que estas demostraron escasa fiabilidad tras numerosas pruebas en las que se probaron distintos ajustes de los parámetros.

Por otro lado, se propone la participación del grupo RobeSafe en el siguiente CARLA Challenge. Una vez conocido el entorno de simulación y las bases y funcionamiento del Challenge, se pretende perfeccionar la aplicación, de manera que sea más robusta y responda mejor ante los distintos retos observados en los escenarios del concurso.

Así mismo, sería interesante la aplicación del controlador depurado e implementado en el simulador CARLA en un vehículo autónomo real.

Bibliografía

[1] R. Siegwart, I. Nourbakhsh. , *Introduction to Autonomous Mobile Robots.*, MIT Press, 2004.

[2] M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte and M. Csorba , *A solution to the simultaneous localization and map building (SLAM) problem*, in IEEE Transactions on Robotics and Automation, vol. 17, no. 3, pp. 229-241, June 2001. doi: 10.1109/70.938381

[3] Hugh Durrant-Whyte and Tim Bailey , *Simultaneous localisation and mapping (slam): Part i the essential algorithms*, IEEE ROBOTICS AND AUTOMATION MAGAZINE, 2:2006, 2006.

[4] A.EliazarandR.Parr., *DP-SLAM:Fast,robustsimultaneouslocalizationandmapping without predetermined landmarks*, 2003.

[5] Jose Guivant and Eduardo Nebot, *Optimization of the simultaneous localization and map building algorithm for real time implementation.*, IEEE Transactions on Robotics and Automation, 17:242?257, 2001.

[6] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, Andrew Ng., *ROS: an open-source Robot Operating System.*

[7] ROS.org, consultado por última vez el 5 de junio de 2019.

[8] https://github.com/cinvesrob/Aria, consultado por última vez el 5 de junio de 2019.

[9] https://www.microsoft.com/en-us/download/details.aspx?id=29081, consultado por última vez el 5 de junio de 2019.

[10] F. Egan, William, *Practical RF System Design.* , Wiley-IEEE Press. ISBN 978-0-471-20023-9., 2003.

[11] U.S. International Trade Commission., *Certain GPS Chips, Associated Software and Systems, and Products Containing Same* , DIANE Publishing. pp. 6-. ISBN 978-1-4578-1632-1.

[12] Toloza, Juan Manuel , *Algoritmos y técnicas de tiempo real para el incremento de la precisión posicional relativa usando receptores GPS estándar* , 2013.

[13] Dr. S. C. Liew , *Electromagnetic Waves.*, Centre for Remote Imaging, Sensing and Processing. 2006.

[14] Martinez Rodriguez, Jairo Alejandro; Vitola Oyaga, Jaime; Sandoval Cantor, Susana del Pilar , *Fundamentos teórico-prácticos del ultrasonido.*, 30 de abril 2007.

[15] Johns Hopkins Bloomberg , *History of Wireless.*,2007.

[16] https://www.quiminet.com/articulos/mida-sus-distancias-con-los-mejores-odometros-2735522.htm, consultado por última vez el 5 de mayo de 2019.

[17] https://es.wikipedia.org/wiki/Codificador_rotatorio#Enlaces_externos , consultado por última vez el 5 de mayo de 2019.

[18] Joydeep Biswas, Manuela M. Veloso , *Depth Camera Based Indoor Mobile Robot Localization and Navigation.*,2012, Robotics and Automation (ICRA), 2012 IEEE International Conference. 1330.

[19] Margrit Betke, Leonid Gurvits , *Mobile Robot Localization using Landmarks.*,1995.

[20] Joel A Hesch, Dimitrios G Kottas, Sean L Bowman and Stergios I Roumeliotis , *Camera-IMU-based localization: Observability analysis and consistency improvement.*,The International Journal of Robotics Research 2014, Vol 33(1) 182?201.

[21] Smith, Randall and Self, Matthew and Cheeseman, Peter. , *Estimating Uncertain Spatial Relationships in Robotics.*,Conference: UAI '86: Proceedings of the Second Annual Conference on Uncertainty in Artificial Intelligence, University of Pennsylvania, Philadelphia, PA, USA, August 8-10, 1986 DOI: 10.1109/ROBOT.1987.1087846

[22] Dieter Fox , Wolfram Burgard , Sebastian Thrun , *Markov Localization for Mobile Robots in Dynamic Environments.*,Journal of Artificial Intelligence Research 11 (1999), pp. 391-427. Submitted 1/99; published 11/99.

[23] Leopoldo Jetto, Sauro Longhi and Giuseppe Venturini , *Development and Experimental Validation of an Adaptive Extended Kalman Filter for the Localization of Mobile Robots.*,IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, VOL. 15, NO. 2, APRIL 1999

[24] Dieter Fox , Wolfram Burgard , Sebastian Thrun , *Markov Localization for Reliable Robot Navigation and People Detection.*, 1999

[25] Musa, ABM , *Advanced Techniques for Mobile Localization and Tracking*, 8 de febrero de 2018.

[26] Latombe, Jean-Claude , *Robot Motion Planning*, New York, 1991.

[27] Thrun, S., Burgard, W. and Fox, D., *Probabilistic Robotics.* Cambridge, Mass: MIT Press. ISBN 0-262-20162-3. OL 3422030M, 2005.

[28] Webster, J. G., Huang, S. and Dissanayake, G. , *Robot Localization: An Introduction.* In Wiley Encyclopedia of Electrical and Electronics Engineering, J. G. Webster (Ed.). doi:10.1002/047134608X.W8318, 2016.

[29] F. N. Sibai, H. Trigui, P. C. Zanini and A. R. Al-Odail, *Evaluation of indoor mobile robot localization techniques*, International Conference on Computer Systems and Industrial Informatics, Sharjah, 2012, pp. 1-6. doi: 10.1109/ICCSII.2012.6454560, 2012.

[30] Thesis by Samuel T. P?ster , *Algorithms for Mobile Robot Localization and Mapping, Incorporating Detailed Noise Modeling and Multi-Scale Feature Extraction.*

[31] Frank Dellaerty, Dieter Foxy, Wolfram Burgardz, Sebastian Thruny , *Monte Carlo Localization for Mobile Robots*, Computer Science Department, Carnegie Mellon University, Pittsburgh PA 15213 zInstitute of Computer Science III, University of Bonn, D-53117 Bonn.

[32] Howie Choset, *Robotic Motion Planning: Bug Algorithms*, Robotics Institute, Carnegie Mellon University (CMU),2005.

[33] S. Quinlan and O. Khatib, *Elastic bands: connecting path planning and control*, Proceedings IEEE International Conference on Robotics and Automation, Atlanta, GA, USA, 1993, pp. 802-807 vol.2. doi: 10.1109/ROBOT.1993.291936

[34] J. Borenstein and Y. Koren , *THE VECTOR FIELD HISTOGRAM - FAST OBSTACLE AVOIDANCE FOR MOBILE ROBOTS*, IEEE Journal of Robotics and Automation Vol 7, No 3, June 1991, pp. 278-288.

[35] Ulrich, I.; Borenstein, J., *VFH+: reliable obstacle avoidance for fast mobile robots*, Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on.

[36] Reid Simmons , *The Curvature-Velocity Method for Local Obstacle Avoidance*,Conference Paper, International Conference on Robotics and Automation, April, 1996.

[37] Ko, N.Y., Simmons, R., *The Lane-Curvature Method for Local Obstacle Avoidance*,in Proceedings of the 1998IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS?98), Victoria, B.C., Canada, October 1998.

[38] Dieter Foxy, Wolfram Burgardy, Sebastian Thrunyz, *The Dynamic Window Approach to Collision Avoidance*,Dept. of Computer Science III, University of Bonn, D-53117 Bonn, Germany, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, P A 15213

[39] O.Brock, O. Khatb, *High-Speed Navigation Using the Global Dynamic Window Approach*, Robotics Laboratory, Dept. of Computer Science Stanford University, Stanford, California.

[40] Schlegel, Christian. , *Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot.*, 594 - 599 vol.1. 10.1109/IROS.1998.724683.

[41] Javier Minguez and Luis Montano, *Nearness Diagram (ND) Navigation: Collision Avoidance in Troublesome Scenarios*, IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, VOL. 20, NO. 1, FEBRUARY 2004.

[42] J. A. Janet, R. C. Luo and M. G. Kay, *The essential visibility graph: an approach to global motion planning for autonomous mobile robots*,Proceedings of 1995 IEEE International Conference on Robotics and Automation, Nagoya, Japan, 1995, pp. 1958-1963 vol.2. doi: 10.1109/ROBOT.1995.526023

[43] P. Bhattacharya and M. L. Gavrilova , *Voronoi diagram in optimal path planning*, 4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007), Glamorgan, 2007, pp. 38-47. doi: 10.1109/ISVD.2007.43

[44] M. M. Zavlanos, M. B. Egerstedt and G. J. Pappas, *Graph-theoretic connectivity control of mobile robot networks*, in Proceedings of the IEEE, vol. 99, no. 9, pp. 1525-1540, Sept. 2011. doi: 10.1109/JPROC.2011.2157884

[45] Ghai, Bhavya and Shukla, A., *Wave Front Method Based Path Planning Algorithm for Mobile Robots*, 2016. 10.1007/978-3-319-30927-9_28.

[46] C. W. Warren, *Global path planning using artificial potential fields*, Proceedings, 1989 International Conference on Robotics and Automation, Scottsdale, AZ, 1989, pp. 316-321 vol.1. doi: 10.1109/ROBOT.1989.100007

[47] Julien Burlet, Olivier Aycard, Thierry Fraichard, *Robust Motion Planning using Markov Decision Processes and Quadtree Decomposition*, Proc. of the IEEE Int. Conf. on Robotics and Automation, Apr 2004, New Orleans, LA (US), France. pp.2820?2825. inria-00182070

[48] Dave Ferguson and Anthony Stentz , *Focussed Processing of MDPs for Path Planning*, Robotics Institute Carnegie Mellon University Pittsburgh, PA, USA

[49] Nicolas Meuleau and Christian Plaunt and David E.Smith and Tristan Smith, *A POMDP for Optimal Motion Planning with Uncertain Dynamics*, Intelligent Systems Division NASA Ames Research Center Moffet Field, California 94035-0001

[50] Yanzhu Du, David Hsu, Hanna Kurniawati, Wee Sun Lee, Sylvie C.W. Ong, Shao Wei Png, *A POMDP Approach to Robot Motion Planning under Uncertainty*, Intelligent Systems Division NASA Ames Research Center Moffet Field, California 94035-0001

[51] Nicolas Meuleau and Christian Plaunt and David E.Smith and Tristan Smith, *A POMDP Approach to Robot Motion Planning under Uncertainty*,Department of Computer Science Stanford University Stanford, CA 94305, USA. Stanford University Stanford, CA 94305, USA. Department of Computer Science National University of Singapore Singapore, 117417, Singapore. Singapore-MIT Alliance for Research and Technology Singapore, 117543, Singapore.

[52] Amalia Foka, Panos Trahanias, *Real-Time Hierarchical POMDPs for Autonomous Robot Navigation*, IJCAI Workshop. Reasoning with Uncertainty in Robotics,Edinburgh, Scotland, 30 July 2005

[53] Y. Abdelrasoul, A. B. S. H. Saman and P. Sebastian, *A quantitative study of tuning ROS gmapping parameters and their effect on performing indoor 2D SLAM*, 2016 2nd IEEE International Symposium on Robotics and Manufacturing Automation (ROMA), Ipoh, 2016, pp. 1-6. doi: 10.1109/ROMA.2016.7847825

[54] HOUGH, P. V. C., *Method and means for recognizing complex patterns.*, U. S. Patent 3, 069 654, December 18 , 1962.

[55] Duda, R. O. and P. E. Hart, *Use of the Hough Transformation to Detect Lines and Curves in Pictures*, Comm. ACM, Vol. 15, pp. 11?15 (January, 1972)

[56] Zadeh, L. A., *Fuzzy sets.*, 1965. Information and Control. 8 (3): 338?353. doi:10.1016/S0019-9958(65)90241-X

[57] Zadeh, L. A., *Fuzzy algorithms.*, 1968. Information and Control. 12 (2): 94?102. doi:10.1016/S0019-9958(68)90211-8

[58] Mamdani, E.H. and S. Assilian, *An experiment in linguistic synthesis with a fuzzy logic controller.*, International Journal of Man-Machine Studies, Vol. 7, No. 1, pp. 1-13, 1975.

[59] L Kavraki, P Svestka, JC Latombe, MH Overmars , *Probabilistic roadmaps for path planning in high-dimensional configuration spaces.*, IEEE Transactions on Robotics and Automation 12 (4), 566-580 , 1996.

[60] R. Craig Coulter, *Implementation of the Pure Pursuit Path Tracking Algorithm.*, The Robotics Institute Carnegie Mellon University, 1992.

[61] Kwakernaak, Huibert and Sivan, Raphael, *Linear Optimal Control Systems.*, First Edition. Wiley-Interscience. ISBN 0-471-51110-2., 1972.

[62] Página oficial de MobileRobots: https://web.archive.org/web/20060422025946/http://www.activrobots.co consultado por última vez el 20 de marzo de 2019.

[63] Página oficial de Simulator: http://www.carla.org, consultado por última vez el 10 de julio de 2019.

[64] Información sobre el simulador *Grand Theft Auto*: https://es.wikipedia.org/wiki/Grand_Theft_Auto, consultado por última vez el 10 de junio de 2019.

[65] Página oficial de CARLA Challenge: https://carlachallenge.org/, consultado por última vez el 10 de julio de 2019.

[66] Alexey Dosovitskiy, German Ros, Felipe Codevilla1, Antonio Lopez, and Vladlen Koltun, *CARLA: An Open Urban Driving Simulator.*

[67] J. Redmon, S. Divvala, R. Girshick and A. Farhadi , *You Only Look Once: Uni?ed, Real-Time Object Detection.*

[68] Albert Soto i Serrano, *YOLO Object Detector for Onboard Driving Images.*

[69] Listado de precios de las plataformas robóticas de MobileRobots: http://www.mobilerobots.com/PDFs/MobileRobotsPriceList.pdf, consultado por última vez el 10 de junio de 2019.

[70] Coulter, R., *Implementation of the Pure Pursuit Path Tracking Algorithm.*, Carnegie Mellon University, Pittsburgh, Pennsylvania, Jan 1990.

[71] Página oficial de AutoWare: http://www.autoware-eu.org/, consultado por última vez el 10 de julio de 2019.

Apéndice A

MANUAL DE USUARIO

En este anexo se describen los pasos a realizar para ejecutar los distintos programas y herramientas necesarias para la reproducción de este proyecto.

A.1 INSTALACIÓN DE ROS

Para comenzar, es necesario la instalación del entorno de desarrollo robótico ROS. Para ello es necesario visitar la página web *http://www.ros.org/* , donde se puede encontrar toda la información necesaria para entender en qué consiste ROS y para qué es utilizado. Los pasos para la instalación se pueden encontrar en el siguiente link (cada versión tiene su propio link y cada una tiene unas restricciones de software propias):

wiki.ros.org/ROS/Installation

Una vez instalado ROS, ha de configurarse el entorno para poder generar código. Es necesario crear un espacio de trabajo, denominado *workspace*, donde se guardarán y compilarán los archivos de los diferentes paquetes creados. En el siguiente link se encuentra un tutorial de creación de un workspace:

http://wiki.ros.org/catkin/Tutorials/create_a_workspace

Para ello, desde un terminal de Ubuntu, se realizarán los siguientes pasos:

1. Creación del espacio de trabajo:

```
mkdir -p tfg_ws/src
```

2. Inicialización del espacio de trabajo:

```
cd tfg_ws/src/
catkin_init_workspace
cd tfg_ws
catkin_make
```

3. Añadir el espacio de trabajo al path por defecto:

```
sudo gedit ~/.bashrc
```
Añadir al final del fichero estas dos lineas:
```
source ~/tfg_ws/devel/setup.bash
 export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:~/tfg_ws/
```

A.2 INSTALACIÓN Y CONFIGURACIÓN DEL SIMULADOR STDR.

Una vez instalado ROS y creado el espacio de trabajo, hay que instalar el simulador STDR, explicado en la sección 2.2.0.2. El simulador puede instalarse desde los repositorios de ROS para Ubuntu con la siguiente instrucción:

```
sudo apt-get install ros-kinetic-stdr-simulator
```

Sin embargo, esta versión tiene unos bugs por solucionar, por lo que es aconsejable instalar el simulador a partir de las fuentes. Para ello puede descargarse desde el siguiente link:

https://github.com/stdr-simulator-rospkg/stdr_simulator

Para probar que la instalación es correcta se puede ejecutar la siguiente instrucción:

roslaunch stdr_launchers server_with_map_and_gui_plus_robot.launch

Deberá aparecer la siguiente ventana mostrando el GUI del simulador, con un mapa y un robot cargados:

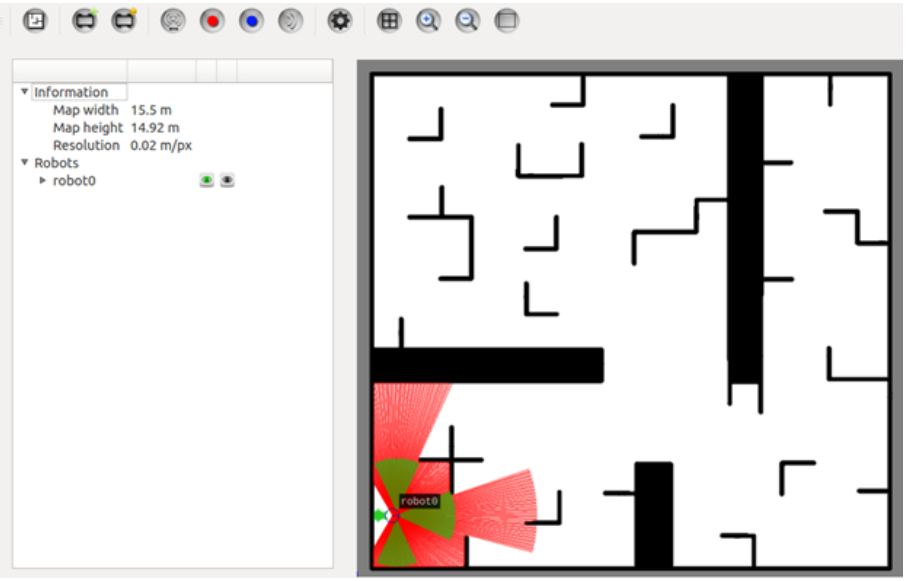

Figura A.1: Ventana del simulador STDR.

A continuación se muestran los pasos para crear un robot AmigoBot con sus sensores asociados (iguales a los que utiliza el robot real), que se utilizará para la simulación de todos los algoritmos desarrollados en el trabajo.

1. Lanzar el simulador, utilizando para ello cualquiera de los ficheros .launch que vienen con la instalación. Por ejemplo, el que carga sólo la GUI y un mapa:

```
roslaunch stdr_launchers server_with_map_and_gui.launch
```

2. Crear un nuevo robot, clicando para ello en *Create robot* (tercer botón de la barra de herramientas superior). Por defecto los robots creados tienen forma circular, aunque se pueden definir formas poligonales utilizando la opción *Footprint* para definir los vértices. En este caso, por simplicidad de diseño se creará el robot circular, con las dimensiones del AmigoBot real, que son 33 x 28 cm (definiendo un radio de 15 cm como aproximación).

3. Añadir un sensor láser, clicando para ello en el botón + en la sección *Lasers*. Esto añadirá un láser llamado *laser_1*. Seleccionando el botón de edición se pueden modificar sus parámetros. Para simular un láser RPLIDAR-A2 deben ajustarse los siguientes valores:
   Number of Rays: 400
   Max distance (m): 8.0
   Min distance (m): 0.15
   Angle span (degrees): 360
   Translation – x(m): 0.09
   Finalmente clicar en *update* y salvar el *laser_1* en la carpeta *laser_sensors* con el nombre *rplidar_a2.xml*.

4. Añadir un sensor de ultrasonidos (sonar), clicando para ello en el botón + de la sección *Sonars*. Esto añadirá un sonar llamado *sonar_1*. Seleccionando el botón de edición se pueden configurar los siguientes parámetros para que simule los sónares del robot AmigoBot:
   Max distance (m): 5.0
   Min distance (m): 0.1
   Con span (degrees): 15
   Frequency: 20
   Translation – x(m): 0.076
   Translation – y(m): 0.1
   Orientation: 90
   Finalmente clicar en *update* y salvar el *sonar_1* en la carpeta *range_sensors* con el nombre *sonar_amigobot.xml*.

5. Crear el anillo de ultrasonidos, añadiendo para ello otros 7 sonares más (de sonar_2 a sonar_8). Para ello, se puede cargar para cada sonar los datos de *sonar_amigobot.xml* y a continuación modificar el parámetro de orientación y posición a los valores mostrados en la figura A.2.

   Quedando el anillo configurado como en la figura A.3.

6. Salvar el robot, con el nombre *amigobot.xml* en la sección *Robot*.

7. Añadir el robot, clicando en el botón *Load robot*, seleccionando el robot que acabamos de crear, y a continuación clicando en la posición del mapa en la que queremos que se posicione el robot. (NOTA: si se obtiene un error al cargar el robot, editar el archivo amigobot.xml y eliminar la sección <kinematics>completa).

Sónar	X	Y	Orientación (º)	Orientación (rad)
0	0.076	0.1	90	1.5708
1	0.125	0.075	41	0.715585
2	0.150	0.03	15	0.261799
3	0.150	-0.03	-15	-0.261799
4	0.125	-0.075	-41	-0.715585
5	0.076	-0.1	-90	-1.5708
6	-0.14	-0.058	-145	-2.53073
7	-0.14	0.058	145	2.53073

Figura A.2: Valores a configurar en el anillo de ultrasonidos.

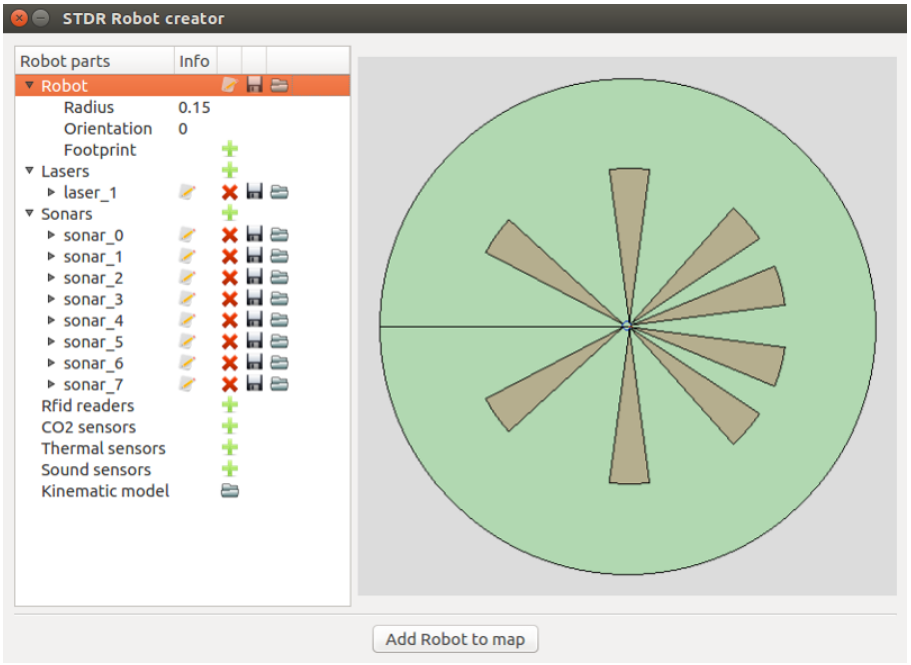

Figura A.3: Anillo de ultrasonidos.

8. Modificar el robot: Una vez generado el robot, el archivo .xml puede ser modificado de forma sencilla. Por coherencia con la numeración de sensores del amigobot se recomienda modificar el archivo para que los sonar vayan desde sonar_0 a sonar_7.

Otro aspecto importante a considerar son los mapas que se cargarán en el simulador durante el desarrollo del trabajo. Estos serán similares a los entornos reales por los que navegarán los robots. Los mapas en ROS vienen definidos por un archivo de imagen (habitualmente en formato png) y un archivo de descripción .yaml. Un ejemplo de archivo de descripción típico es el siguiente:

`image: sparse_obstacles.png`, indica el archivo de imagen a cargar.
`resolution: 0.02` , en metros por píxel.
`origin: [0.0, 0.0, 0.0]` , origen del mapa (x, y, orientación).
`occupied_thresh: 0.6` , los píxeles con más de esta ocupación en la escala de grises se consideran ocupados.
`free_thresh: 0.3` , los píxeles por debajo de esta ocupación en la escala de grises se consideran libres.
`negate: 0` , si está a 0, los píxeles que tienden a blanco son libres y los negros ocupados. Si está a 1 se invierte esta consideración.

A.3 Herramientas para trabajar en ROS.

En el apartado 2.2.0.1, se explican brevemente algunos de los conceptos básicos para comenzar a trabajar con ROS.

Aquí se nombran algunas de las herramientas o comandos más útiles para trabajar con ROS.

A.3.1 Comandos para trabajar con nodos:

Para monitorizar y obtener información de los nodos que están en ejecución en un determinado momento, se dispone de la herramienta *rosnode*.

A modo de ejemplo, mientras ejecutamos el simulador con la prueba de instalación, podemos hacer uso de esta herramienta en otro terminal. Los comandos más utilizados son:

- **rosnode list**: devuelve una lista de los nodos que están activos.
- **rosnode info *nombre_nodo***: devuelve información de ese nodo, como sus publicaciones, sus subscripciones, los servicios que ofrece y sus conexiones a los demás nodos.
- **rosnode kill *nombre_nodo***: elimina un nodo de la ejecución.

Una forma cómoda de visualizar todos los nodos que se encuentran en ejecución en un determinado momento, es ejecutando la herramienta rqt en un terminal y a continuación cargando el plugin *plugin/introspection/node_graph* y la vista *nodes only*, obteniéndose un grafo como el que se muestra en la figura A.4.

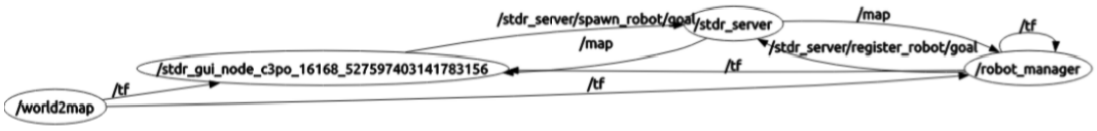

Figura A.4: Herramienta rqt: Node_graph.

A.3.2 Comandos para trabajar con topics:

La herramienta rostopic nos permite inspeccionar estos canales de comunicación. Sus comandos más útiles son los siguientes:

- **rostopic list**: lista todos los topics activos.
- **rostopic info *nombre_topic*** : muestra la información del topic indicado como el tipo de dato que transmite, nodos que están publicando en el topic y nodos que están leyendo de él.
- **rostopic type *nombre_topic*** : muestra la información del tipo de dato que se transmite por el topic.
- **rostopic echo *nombre_topic*** : crea un *subscriber* desde línea de comandos y vuelca a pantalla los datos que se están transmitiendo por ese topic.
- **rostopic pub *nombre_topic tipo de dato datos***: Permite publicar datos en un topic desde consola generando un *publisher*.

Al igual que con los nodos, se puede realizar una inspección visual de los topics mediante la herramienta rqt, en este caso seleccionando el plugin *plugin/introspection/node_graph* y la vista *node/topics (active)*, que muestra en elipses los nodos y en cuadrados los topics con los que se comunican, como se observa en la figura A.5.

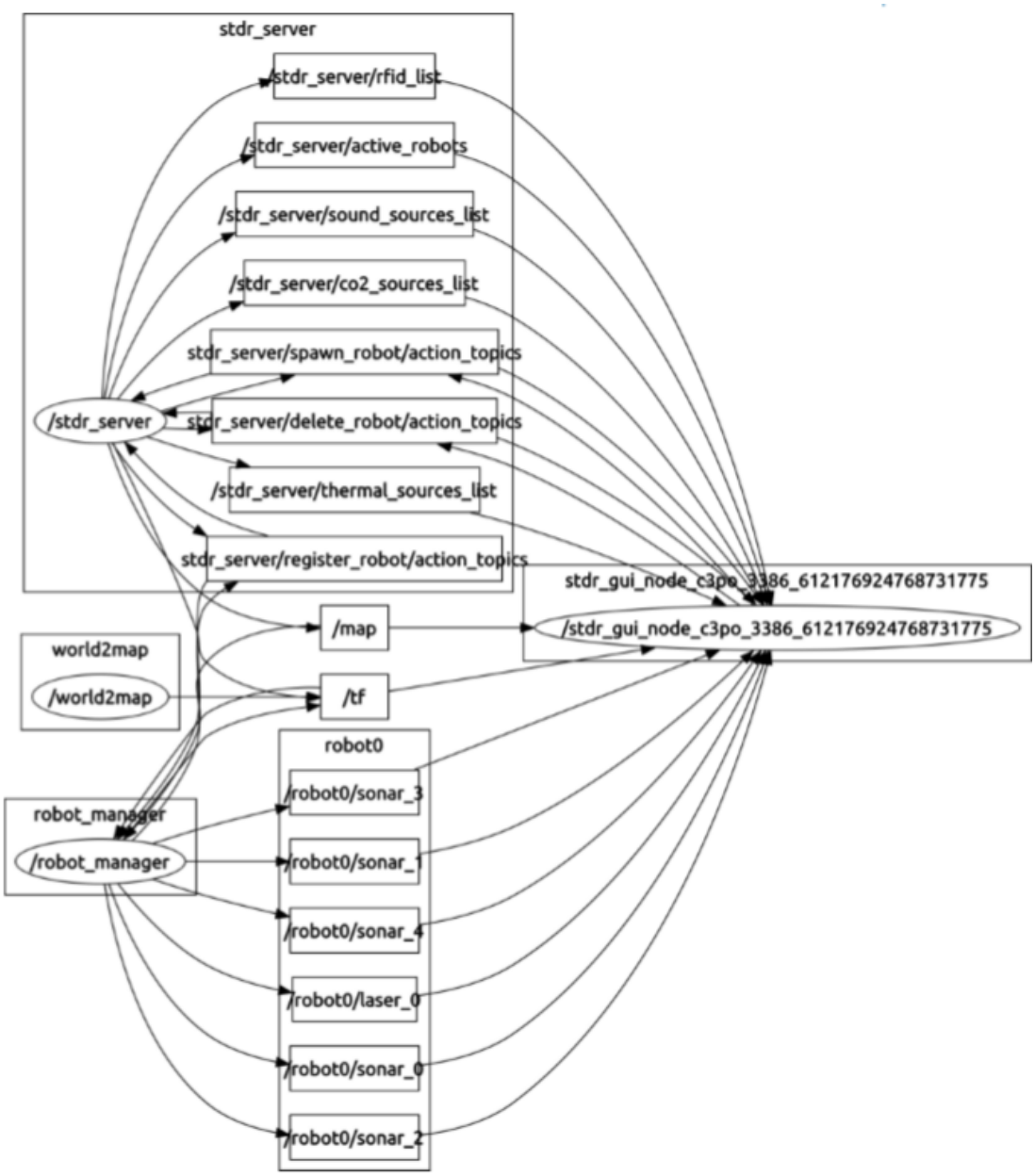

Figura A.5: Herramienta rqt: Node_graph con topics.

A.3.3 TF (Transformadas).

/tf es un topic especial de ROS que está siempre presente y sirve para relacionar los distintos marcos de coordenadas que existen en el sistema. Es gestionado por un paquete llamado TF que incluye múltiples utilidades para trabajar con los marcos de coordenadas y gestionar sus transformaciones, y que se verá en detalle en prácticas posteriores. La relación entre los sistemas de coordenadas, incluso cuando es constante a lo largo del tiempo, ha de ser publicada de forma continua en el topic /tf para que el resto de nodos pueda utilizar esta información.

Usualmente, cada topic contiene información referida a un sistema de coordenadas. Este sistema de coordenadas se encuentra indicado explícitamente en la cabecera del tipo de datos (campo frame_id) con excepción de algunos tipos de datos que no están referidos a ningún marco de referencia y carecen de ese campo (por ejemplo los datos de tipo entero).

Para visualizar el árbol de transformadas se puede utilizar la herramienta rqt, plugin (plugins/visualization/TF Tree) el cual mostrará la relación entre los distintos marcos de coordenadas así como quién genera cada transformada y con qué frecuencia la actualiza.

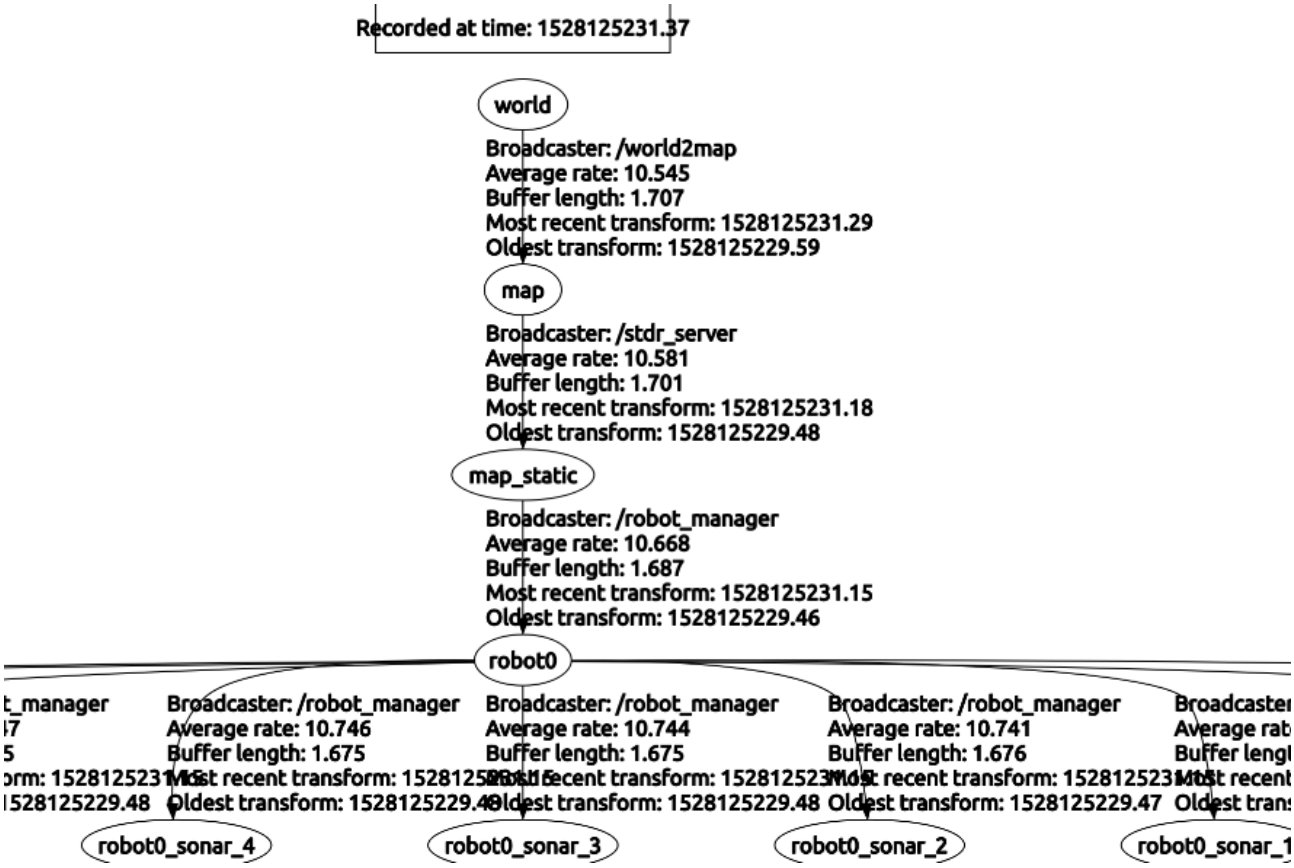

Figura A.6: Herramienta rqt: TF Tree.

En el grafo mostrado en la figura A.6 se puede comprobar que existe un marco base de coordenadas llamado *world* relacionado con otro marco de coordenadas llamado *map*. Esta relación la realiza el nodo *world2map* el cual publica una transformada estática entre ambos. A su vez el marco *map* está relacionado por una transformada estática enviada por el simulador (stdr_server) con el marco de coordenadas map_static. Los marcos de coordenadas del mapa (map_static) y el robot (robot0) están relacionados por la odometría del mismo, relación que cambia conforme se mueve el robot y está publicada por el simulador (nodo robot_manager). También existen una serie de transformadas entre el robot y sus diferentes sensores, que son publicadas por el simulador.

Hay que tener en cuenta una restricción importante a la hora de crear un árbol de transformadas en ROS: un marco de referencia puede tener varios *hijos* pero solamente un único *padre*. De tal forma que siempre existirá un único elemento raíz (comúnmente world o map en sistemas de robots móviles) del que irán colgando los distintos elementos que existen en el entorno (distintos robots, sensores del entorno, etc.).

A.4 Ejecución de un programa.

Para la ejecución de un programa en ROS existen dos maneras: el comando `rosrun` y el comando `roslaunch`.

- **Rosrun**: este comando permite ejecutar un único nodo siempre que exista un nodo *roscore* activo. Su formato es el siguiente:

  ```
  rosrun nombre_paquete nombre_nodo argumentos
  ```

  Como ejemplo podemos ejecutar un nodo que nos permite teleoperar el robot en el simulador:

  ```
  rosrun teleop_twist_keyboard teleop_twist_keyboard.py
  cmd_vel:=/robot0/cmd_vel _speed:=0.3 _turn:=0.5
  ```

  Esta instrucción ejecuta el nodo *teleop_twist_keyboard.py* que se encuentra dentro del paquete *teleop_twist_keyboard.py* . Se realiza una redirección de topics cambiando el topic en el que escribe por defecto (cmd_vel) al del robot en simulación (/robot0/cmd_vel). También se modifican dos parámetros (speed y turn) limitando la velocidad máxima comandada a 0.3 m/s de velocidad lineal y 0.5 rad/s de velocidad angular.

- **Roslaunch**: dada la complejidad que puede conllevar la ejecución de ROS, se utilizan los comandos roslaunch a modo de script para lanzar distintos nodos en una única instrucción. Un detalle importante es que el archivo .launch ejecuta automáticamente un nodo *roscore* en caso de que el mismo no se encuentre activo. El formato es el siguiente:

  ```
  roslaunch nombre_paquete fichero.launch
  ```

  Para que un archivo pueda ser ejecutado ha de encontrarse dentro de la carpeta /launch del paquete correspondiente. El fichero roslaunch puede ejecutar distintos nodos, incluir ficheros de configuración de parámetros e incluso incluir otros ficheros .launch. Posteriormente se detallará con un ejemplo el formato de este tipo de archivos.

A.5 Creación de un fichero .launch

Los ficheros .launch permiten lanzar simultáneamente un conjunto de nodos. A modo de ejemplo, se va a crear a continuación un archivo *amigobot.launch*, basado en el fichero *server_with_map_and_gui.launch* que permitirá lanzar el simulador con el robot amigobot que se ha creado en un apartado previo.

Para poder ejecutar el archivo correctamente ha de encontrarse dentro de la carpeta /launch de un paquete, por lo que creamos el directorio *~/tfg_ws/src/simulacion/launch/* y dentro de él el archivo *amigobot.launch* de la siguiente manera:

```
<launch>

<include file="$(find stdr_robot)/launch/robot_manager.launch" />

//Inclusión de otro archivo .launch el cual lanza un nodo que
   proporciona la descripción del robot a ROS

<node type="stdr_server_node" pkg="stdr_server" name="stdr_server"
   output="screen" args="$(find stdr_resources)/maps/robocup.yaml"/>

// Nodo que lanza el mapa con el que trabajarán ROS y STDR. Se pasa
   la descripción del mapa (.yaml) como parámetro clásico (campo args
   ). Se modifica el mapa respecto al original para representar un
   mapa de tipo laberinto.

<node pkg="tf" type="static_transform_publisher" name="world2map"
   args="0 0 0 0 0 0  world map 100" />

//  Nodo que genera la relación entre los marcos de coordenadas map (
   mapa del simulador) y world (marco de coordenadas raíz). Se indica
    esta relación como argumentos del nodo y que siguen el formato "x
    y z yaw pitch roll frame_origen frame_destino frecuencia"

<include file="$(find stdr_gui)/launch/stdr_gui.launch"/>

//  Inclusión de archivo .launch que lanza la interfaz gráfica del
   simulador.

 <node pkg="stdr_robot" type="robot_handler" name="$(anon robot_spawn
    )" args="add $(find stdr_resources)/resources/robots/ amigobot.
    xml 4 8 0" />

 //  Nodo que carga el robot Amigobot creado en la posición (4,8,0)
```

```
</launch>
```

Para lanzar esta simulación, ejecutamos el fichero .launch anterior del siguiente modo:

```
roslaunch simulacion amigobot.launch
```

y observaremos la pantalla del simulador como en la figura A.7.

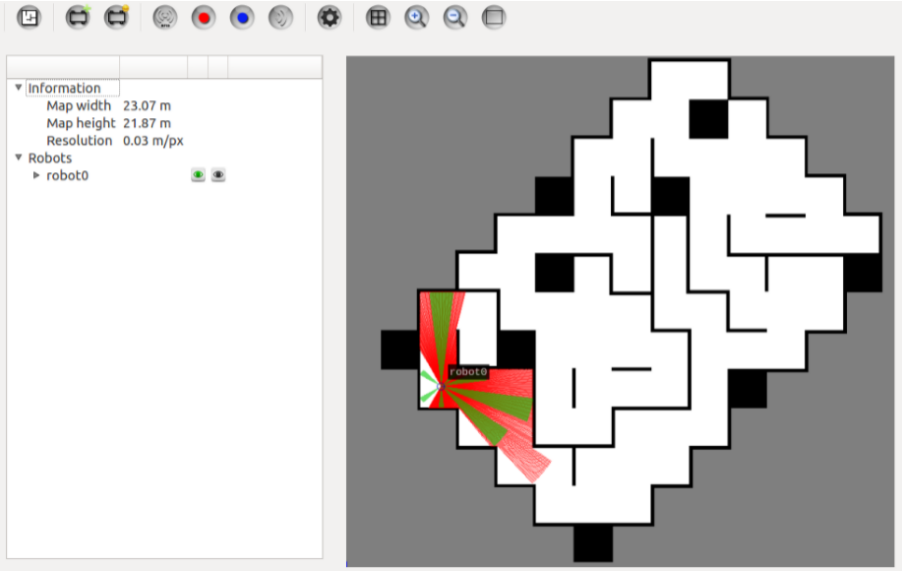

Figura A.7: Ventana de simualción ejecutada por amigobot.launch.

A.6 PROGRAMACIÓN MATLAB-ROS.

En esta sección se describe como establecer la conexión MATLAB-ROS, así como los pasos para ejecutar las aplicaciones desarrolladas en este trabajo. El procedimiento es similar en todas ellas.

A.6.1 Configuración de la red.

Si estamos ejecutando Matlab y ROS en máquinas distintas, ha de configurarse la red de tal manera que exista conexión entre ellas. Para ello, se recomienda comprobar que todos los ordenadores que se están empleando en el sistema (incluida la máquina virtual) se encuentran en la misma red y todos son visibles a todos, para ello se pueden emplear herramienta ping para comprobar la visibilidad.

El Máster de ROS (roscore) se ejecutará únicamente en una de las máquinas: en este proyecto se ejecutó bien en la máquina virtual con el simulador, bien en el robot real. El resto de nodos deben conocer la dirección IP de la máquina en la que se ejecuta el máster.

Para conocer la dirección IP actual de la máquina virtual debe abrirse un terminal y ejecutarse el comando `ifconfig`.

Para configurar correctamente las variables de entorno con la dirección IP actual de la máquina virtual debe editarse el fichero .bashrc:

```
cd
gedit .bashrc
Al final del fichero hay que añadir las siguientes líneas con la IP obtenida anteriormente:
export ROS_MASTER_URI=http://IP_ROSMASTER_MACHINE:11311
export ROS_IP=IP_LOCAL_MACHINE
```

A.6.2 Ejecución de las aplicaciones.

A.6.2.1 Simulación.

Para ejecutar la aplicación en cuestión o programa cliente necesitaremos lanzar el simulador STDR en Ubuntu:

```
roslaunch simulacion amigobot.launch
```

y una vez lanzado, ejecutar el programa en Matlab. El nombre y directorio de estos programas se detalla en el anexo *Planos*, en la sección C.2. Primero, habrá que ejecutar *conectar.m* para establecer la conexión entre las máquinas. Tras esto, hay que correr el programa *ini_simulador.m*, el cual declara los subscribers y publishers y define la periodicidad del timer. Una vez hecha la inicialización, ya podemos ejecutar la aplicación que se desee.

Además, en las aplicaciones de mapeado y localización será necesario teleoperar el robot. Para ello hay que ejecutar el siguiente nodo en Ubuntu, que nos permite teleoperar el robot en el simulador:

```
Instalación del paquete: sudo apt-get install ros-indigo-teleop-twist-keyboard
Ejecución del nodo:
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
cmd_vel:=/robot0/cmd_vel _speed:=0.3 _turn:=0.5
```

El algoritmo de SLAM con *gmapping* es el único cuya ejecución difiere de las otras aplicaciones, ya que corre completamente dentro de Ubuntu. Para reproducirlo hay que seguir los siguientes pasos:

1. Instalación del paquete *gmapping* desde los repositorios:

   ```
   sudo apt–get install ros–indigo–gmapping
   ```

2. Lanzar el fichero *gmappingSimulator.launch*

3. Guardar el mapa generado con la siguiente instrucción:

   ```
   rosrun map_server map_saver map:=gmapping_map –f nombre_archivo
   ```

A.6.2.2 Robot Real.

Al robot Amigobot se le ha incorporado como procesador externo una tarjeta Raspberry PI con Linux y ROS, a la cual se ha conectado un sensor láser RPLIDAR-A2. La Raspberry PI de cada robot tiene configurada una dirección IP fija para poder conectarse en red con cualquier otro ordenador externo.

En el arranque del robot, el sistema operativo ejecuta un script que configura y lanza de forma automática un Master de ROS (roscore), así como los drivers de ROS necesarios para controlar el robot y acceder a los datos de los sensores (p2os y rplidar).

Primero habrá que configurar correctamente el sistema distribuido: variables de entorno y funcion *rosinit* en *conectar.m*.

Después hay que ejecutar *ini_amigobot.m*. Las diferencias de este con *ini_simulador.m* son el nombre de los topics del robot a los que nos suscribimos y el publisher necesario para habilitar los motores antes de que empiece a desplazarse.

El resto de scripts solo presentan pequeñas diferencias para simulación y para los robots reales, explicados en la sección correspondiente a cada aplicación dentro del capítulo Desarrollo.

Igualmente, para las aplicaciones de mapeado y localización será necesario lanzar el nodo de teleoperación desde Ubuntu.

Para el algoritmo de SLAM con *gmapping* hay que proceder igual que en simulación, lanzando el nodo gmapping_RealRobot.launch.

A.7 Ejecución del agente Robesafe.py en CARLA.

Dentro del último apartado del capítulo de Desarrollo, en la sección 3.7.4, se detallaron los pasos a seguir para la instalación y puesta en marcha del simulador CARLA, así como para la creación del agente Robesafe.py.

En este manual, se describirán los pasos para lanzar el simulador y el agente.

1. Abrir el simulador en un terminal:

```
cd /home/user/UnrealEngine_4.21/carla/Dist/0.9.5/LinuxNoEditor

./CarlaUE4.sh /Game/Carla/Maps/Town01 -benchmark -fps=20 -
   windowed
```

2. Lanzar el agente con el evaluador en otro terminal terminal:

```
cd /home/user/UnrealEngine_4.21/scenario_runner

python srunner/challenge/challenge_evaluator_routes.py --
   scenarios=srunner/challenge/all_towns_traffic_scenarios1_3_4.
   json --routes=srunner/challenge/routes_training.xml --agent=/
   workspace/team_code/RobesafeAgent.py
```

Apéndice B

PLIEGO DE CONDICIONES

B.1 Requisitos de Hardware

- PC compatible. Cuanta mayor memoria RAM y capacidad de cómputo posea, mejor funcionamiento tendrán los algoritmos.
- Es aconsejable tarjeta de aceleración gráfica si se desea trabajar con el simulador, puesto que el consumo de recursos es alto.
- Plataforma robótica Amigobot y drivers necesarios para su control en ROS.
- Plataforma robótica Seekur y drivers necesarios para su control en ROS.
- Láser RP LIDAR A2 con los drivers compatibles en el entorno de desarrollo ROS.
- Láser SICK LMS151 con los drivers compatibles en el entorno de desarrollo ROS.
- Raspberry Pi.

B.2 Requisitos de Software

- Sistema operativo GNU/Linux, distribución Ubuntu, las pruebas y la programación de los algoritmos en ROS han sido realizadas en la versión 16.04.6 LTS.
- Sistema operativo Windows, las pruebas y la programación de los algoritmos han sido realizadas en la versión Windows 10.
- Software de desarrollo robótico: ROS.
- Software de desarrollo: MATLAB. Robotic System Toolbox.

Apéndice C

PLANOS

En este capítulo se recogen los esquemáticos de los láseres integrados en las plataformas robóticas utilizadas así como una estructura de los ejecutables y programas usados durante el proyecto.

C.1 Esquemáticos.

En esta sección se muestran los esquemáticos de los láseres RP LIDAR A2 y SICK LMS151.

LÁSER RP LIDAR A2

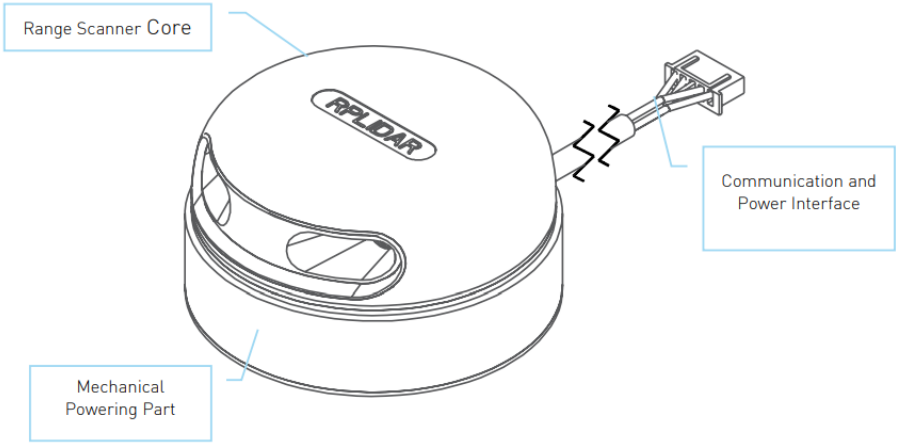

Figura C.1: Composición del sistema RPLIDAR.

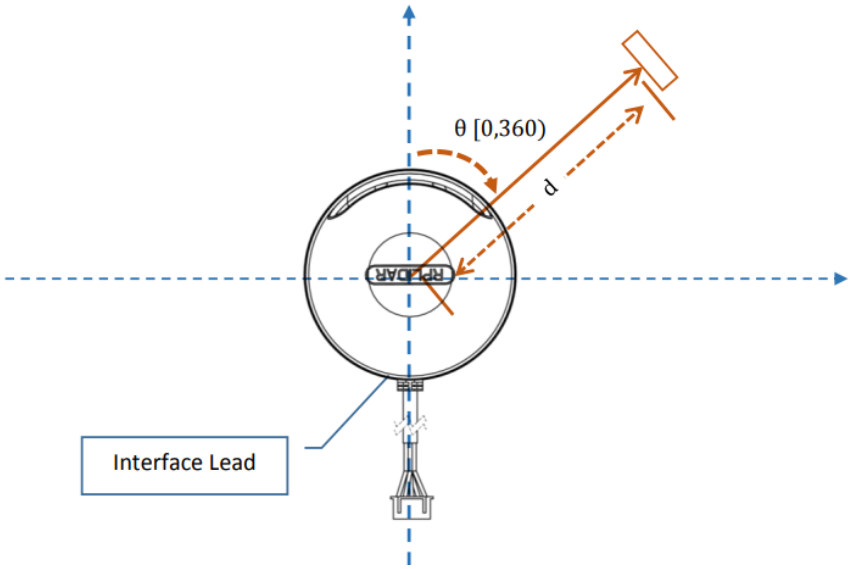

Figura C.2: Definición del sistema de coordenadas para el escaneo de datos.

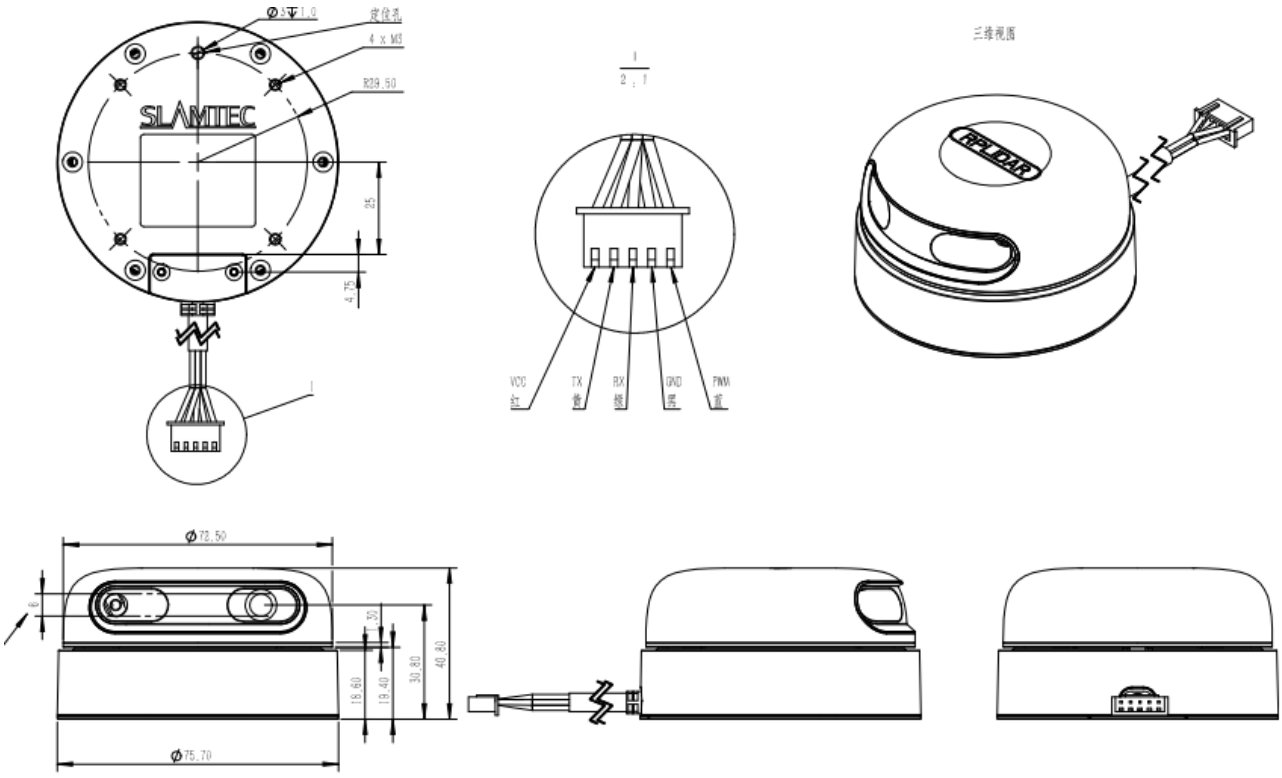

Figura C.3: Dimensiones mecánicas.

LÁSER SICK LMS151

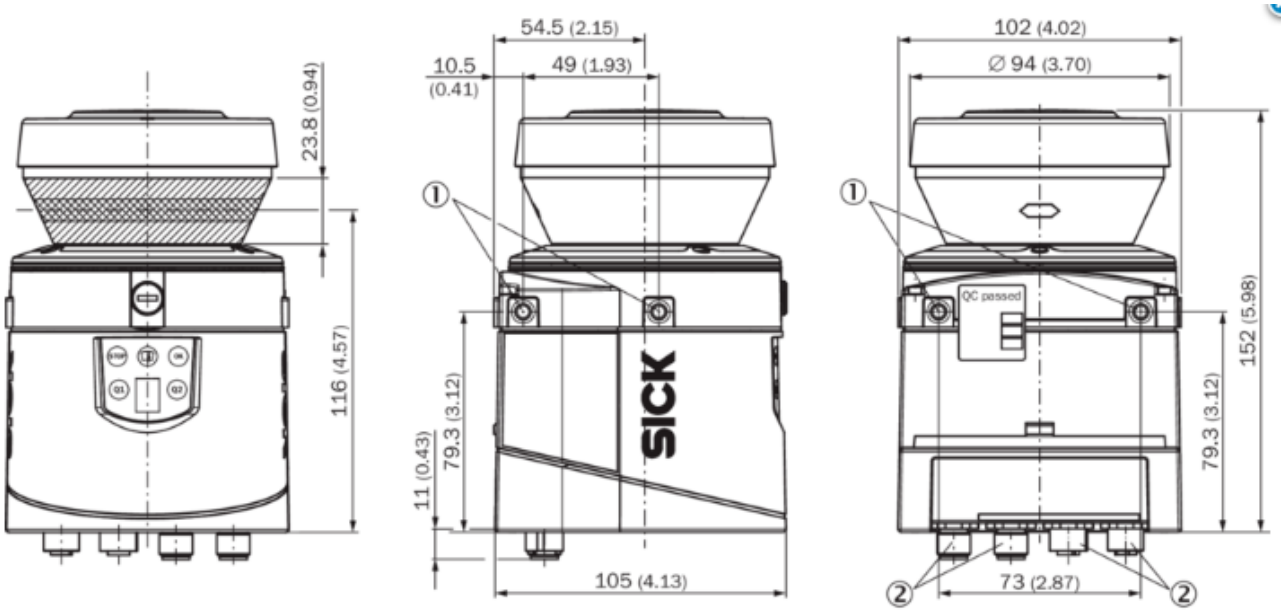

Figura C.4: Dimensiones mecánicas.

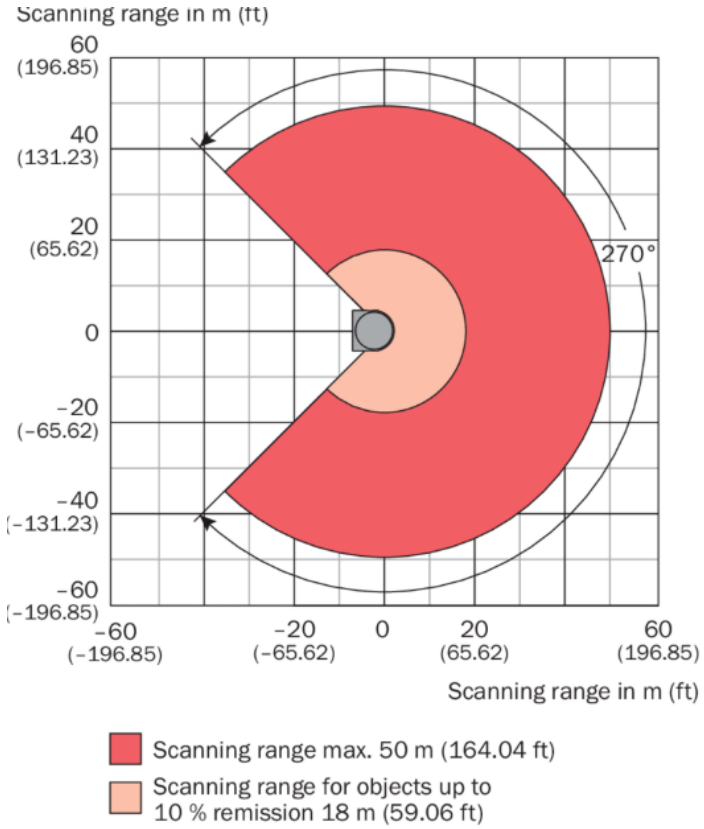

Figura C.5: Diagrama del área de trabajo.

C.2 Códigos de los programas.

En esta sección se citan los ejecutables y programas más usados en este proyecto. Todos ellos se encuentran dentro de la carpeta `PROGRAMAS`. Dentro de la cual se encuentran los siguientes directorios:

- PRUEBAS INICIALES: En esta carpeta encontramos todos los códigos utilizados para inicializar la arquitectura de trabajo y establecer la conexión entre las distintas máquinas.
  - `conectar`: establece la comunicación con la máquina en la que se ejecuta ROS.
  - `Desconectar`: corta la comunicación con la máquina en la que se ejecuta ROS.
  - `amigobot.launch`: fichero .launch que lanza el simulador STDR con el robot Amigobot configurado.
  - `ini_simulador`: declaración de subscribers y publishers para el trabajo en simulación, con los correspondientes mensajes a publicar. Se define la periodicidad del bucle y con el último bucle se comprueba la recepción de mensajes.
  - `ini_amigobot`: declaración de subscribers y publishers para el trabajo con los robots reales, con los correspondientes mensajes a publicar. Se define la periodicidad del bucle y con el último bucle se comprueba la recepción de mensajes.
  - `lee_sensores`: prueba de la correcta comunicación entre las máquinas mediante subscribers. Permite comprobar la satisfactoria lectura de los sensores.
  - `avanzar`: prueba de la correcta comunicación entre las máquinas mediante publishers. Hace avanzar al robot una distancia *d*, comprobando el control de la velocidad lineal.
  - `girar`: con este programa se comprueba la correcta obtención de la orientación del robot transformándolo a ángulos de Euler. Hace girar al robot un ángulo *th*.
- MAPEADO Y SLAM: En esta carpeta encontramos todos los códigos utilizados para implementar las aplicaciones de mapeado. Además de los programas de conexión, desconexión e inicialización de las plataformas, se encuentran:
  - `OfflineSLAM`: algoritmo de SLAM offline. Los ficheros *.bag* adjuntos contienen las medidas del láser durante un recorido del robot por el pasillo.
  - `OnlineSLAM`: algoritmo de SLAM online.
  - `MappingWithKnownPoses_Simulator`: algoritmo de mapeado en simulación asumiendo posiciones conocidas.
  - `tf_repub`: nodo que coge la información que sí nos interesa del nodo *tf*; esta es, la relación entre los marcos de coordenadas del láser y los sonar respecto al robot.
  - `aux_files.launch`: nodo que configura un error no nulo en la odometría del robot para el simulador.
  - `gmapping_Simulator.launch`: fichero que lanza el nodo gmapping para simulación.

  - `tf_RealRobot.launch`: fichero que lanza el nodo gmapping para los robots reales.
  - `gmapping_mapreal1.pmg` y `gmapping_mapreal1.yaml`: ficheros de representación del mapa del pasillo obtenido con *gmapping* mediante el nodo *map_saver*.

- LOCALIZACIÓN: En esta carpeta encontramos todos los códigos utilizados para implementar las aplicaciones de localización con el método de Monte Carlo. Además de los programas de conexión, desconexión e inicialización de las plataformas, se encuentran:

  - `MonteCarloLocalization_Simulator`: algoritmo de AMCL para simulación. El mapa de partida es *map_pasillo.mat*.
  - `MonteCarloLocalization_RealRobot`: algoritmo de AMCL para robots reales. El mapa de partida es *map_pasillo.mat*.

- EVITACIÓN DE OBSTÁCULOS: En esta carpeta encontramos todos los códigos utilizados para implementar la aplicación de evitación de obstáculos con el algoritmo *Vecto Field Histogram*. Además de los programas de conexión, desconexión e inicialización de las plataformas, se encuentran:

  - `VFH_SIMU`: algoritmo de VFH para simulación.
  - `VFH_ROBOT`: algoritmo de VFH para las plataformas robóticas reales.

- SEGUIMIENTO DE PASILLOS: En esta carpeta encontramos todos los códigos utilizados para implementar la aplicación de seguimiento de pasillos mediante lógica borrosa y la Transformada de Hough. Además de los programas de conexión, desconexión e inicialización de las plataformas, se encuentran:

  - `SEGUIMIENTO_PASILLO`: programa que implementa el seguimiento de pasillos para el Amigobot.
  - `fuzzy_mandami.fiz`: archivo creado con la herramienta *Fuzzy Logic Designer* que implementa el controlador borroso.

- PLANIFICACIÓN GLOBAL: En esta carpeta encontramos todos los códigos utilizados para implementar la aplicación de planificación global con el método Probabilistic Roadmap. Además de los programas de conexión, desconexión e inicialización de las plataformas, se encuentran:

  - `PathFollowingControllerSimulation`: primera prueba del algoritmo PRM en simulación.
  - `VFH_AMCL_PRM`:programa que implementa la aplicación global que unifica la localización, planificación local y planificación global. El mapa de partida para la aplicación es *mapa_pasillo.mat*.

- CARLA_CHALLENGE: En esta carpeta se encuentran los códigos relacionados con el CARLA Challenge.
  - `spline_node.c`: este es el nodo principal en el que se programa la generación de trayectoria mediante interpolación de tipo spline y el controlador LQR con predicción.
  - `object_detection.c`: nodo encargado de recibir y procesar los datos provenientes del lidar y las cámaras con el fin de detectar semáforos y objetos.
  - `RobesafeAgent.py`: agente enviado al concurso CARLA Challenge, el cual lanza todos los nodos, lleva a cabo todas las inicializaciones, recoge y publica en topics los datos de los sensores y envía los comandos de aceleración y dirección al vehículo de CARLA.

C.3 Vídeos de ejemplo de las aplicaciones implementadas.

En esta sección se adjuntan los enlaces de acceso a los vídeos de ejemplo de cada una de las aplicaciones implementadas en las distintas plataformas.

- MAPEADO Y SLAM:
  - SLAM simulación: https://youtu.be/B4qeKiz1X0Y
  - SLAM Amigobot: https://youtu.be/Zvuo-vxu2uk
  - SLAM Seekur: https://youtu.be/vI954WPLx_c
  - Gmapping Simulación: https://youtu.be/zoBS9cTNXYI
  - Gmapping Amigobot: https://youtu.be/gnqIWsKz0w0
  - Gmapping Seekur: https://youtu.be/Hifvo1zBjBc
- LOCALIZACIÓN:
  - Localización local simulación: https://youtu.be/gWOCvxbT6x8
  - Localización local Amigobot: https://youtu.be/aM-pyu2v1m0
  - Localización local Seekur: https://youtu.be/tocrLqWc2dk
  - Localización global Simulación: https://youtu.be/nMVGtXPHA0M
  - Localización global Amigobot: https://youtu.be/SG-YI37aRUE
  - Localización global Seekur: https://youtu.be/pW_aPEd39Cg
- PLANIFICACIÓN LOCAL:
  - VFH simulación: https://youtu.be/UFGndMSTUBY
  - VFH Amigobot: https://youtu.be/G1yhTEDKJRc
  - VFH Seekur: https://youtu.be/lM0sO38b2U4
  - Seguimiento de pasillos simulación: https://youtu.be/L_CGbM_csZs

  - Seguimiento de pasillos Amigobot: https://youtu.be/xacnA6kDaeY

- PLANIFICACIÓN GLOBAL:
  - PRM simulación: https://youtu.be/qx5X3QKosHc
  - Aplicación completa (AMCL+VFH+PRM+PurePursuit) Amigobot: https://youtu.be/8aMScDaFPeU

- CARLA CHALLENGE:
  - https://youtu.be/22icW48Z8Vs
  - https://youtu.be/gncjfWjIQ0s

Apéndice D

Presupuesto

En este anexo se detalla el presupuesto de ejecución del proyecto. Se compone de las siguientes partes:

- Coste por uso de equipos y Software.
- Coste de personal.
- Gastos generales, beneficio industrial e I.V.A.

La duración del proyecto ha sido de nueve meses teniendo en cuenta la ampliación con la parte de CARLA. En este periodo de tiempo se incluye los tiempos de aprendizaje y de ejecución, y ha representado el trabajo de un Ingeniero Industrial a tiempo parcial.

D.1 Coste por uso de Hardware y Software.

- Ordenador portátil HP ProBook 470 G5, con procesadorIntel® Core i7-8550U, 8GB de RAM, Disco duro SATA de 1TB 5400rpm + SSD 256GB, Gráficos NVIDIA® GeForce® 930MX (2GB DDR3).
- Software de libre distribución: sistema operativo GNU/LINUX, plataforma de desarrollo ROS.
- Software matemático MATLAB.
- Sistema operativo Windows 10.
- Plataforma robótica Amigobot y drivers necesarios para su control en ROS.
- Plataforma robótica Seekur y drivers necesarios para su control en ROS.
- Láser RP LIDAR A2 con los drivers compatibles en el entorno de desarrollo ROS.
- Láser SICK LMS151 con los drivers compatibles en el entorno de desarrollo ROS.
- Raspberry Pi 3.

Concepto	Precio Unitario (€)	Amortización	Uso	Total(€)
Portátil HP	1300	5 años	9 meses	195
MATLAB	250	1 año	9 meses	187.5
SO: Windows 10	19.95	Ilimitada	9 meses	19.95
Amigobot	1998.98	8 años	3 meses	62.46
Seekur	33833.3	10 años	1 mes	281.9
RP LIDAR A2	337.59	5 años	3 meses	16.8
SICK LMS151	3998.8	5 años	1 mes	66.6
Raspberry Pi 3	31.23	4 años	3 meses	1.95
TOTAL:				**832.16**

Tabla D.1: Presupuesto de HW y SW.

D.2 Coste de personal.

A continuación se detallan los sueldos base por hora trabajada de las personas que serían necesarias para realizar este trabajo, así como el coste que producirían las mismas. Se considera que se ha trabajado 20 horas semanales y se ha estimado que el mes tiene una media de 22 días laborables. Por lo tanto, para un total de 9 meses, el coste de personal ha sido:

Concepto	Horas	Coste por hora (€)	Total(€)
Ingeniero Industrial	792	20	15840
TOTAL:			**15840**

Tabla D.2: Presupuesto de personal.

D.3 Presupuesto total

El coste total de personal, equipos y software se denomina Presupuesto de Ejecución de Materiral (PEM).

Concepto de costes	Precio(€)
Equipos Hardware y Software	832.16
Personal	15840
TOTAL:	**16672.16**

Tabla D.3: Presupuesto de ejecución de material.

En el presupuesto de Ejecución por Contrata se incluye el coste de ejecución material (PEM) junto con los gastos generales, el beneficio industrial y los honorarios de dirección y redacción.

Concepto	Valor (% PEM)	Precio(€)
PEM	100	16672.16
Gastos generales y beneficio industrial	15	2500.824
Honorarios por redacción	7	1167.1
Honorarios por dirección	7	1167.1
TOTAL:		**21507.1**

Tabla D.4: Presupuesto de ejecución por contrata.

Tras haber expuesto todos los gastos generados en el proyecto, reflejados en la tabla D.4, al presupuesto total hay que añadirle el IVA.

Concepto	Precio(€)
Presupuesto de ejecución por contrata	21507.1
IVA (21 %)	4516.49
TOTAL:	**26023.59**

Tabla D.5: Presupuesto total del proyecto.

El presupuesto total del proyecto asciende a una cantidad aproximada de veintiséis mil veinte euros a fecha de mayo del año 2019.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá